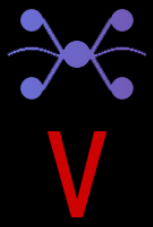
Synapse_IDE



for Psychiatry Professionals

SynapseIDE

A Programmable Digital Psychiatry Workbench

Digital psychiatry, multi-model AI orchestration,
measurement-based care, and embedded clinical tools

Mustafa Raşit Şahin, PhD

Department of City and Regional Planning,
Middle East Technical University

Synapse_IDE for Clinical Psychiatry Professionals

SynapseIDE

A Programmable Digital Psychiatry Workbench

Multi-Model AI Orchestration, Measurement-Based Care, and Embedded Clinical Tools

---

Mustafa Raşit Şahin, PhD

Department of City and Regional Planning
Middle East Technical University

rsahin@metu.edu.tr

ORCID: 0009-0001-4809-3950

Draft (Preprint) Monograph

Abstract

Psychiatric care is increasingly constrained by intensive documentation requirements, formalised measurement-based care (MBC) protocols, and institutional expectations that clinicians integrate digital tools into routine practice. In most health systems, however, documentation workflows, MBC instruments, guideline processes, and generic large language model (LLM) chat interfaces remain only loosely coupled. Clinicians must repeatedly switch between electronic health record (EHR) screens, web calculators for rating scales, local spreadsheets, and ad hoc AI tools that neither encode psychiatry-specific structure nor provide transparent control over prompts, parameters, or data flows. From a cognitive perspective, this fragmentation increases working-memory load, disrupts clinical reasoning, and obscures the mapping from *observations* to *inferences* to *documented decisions*. From a methodological perspective, it makes it difficult to define programmable, auditable AI workflows that can be evaluated, governed, and iteratively improved.

SynapseCore is a web-based, React/TypeScript workbench that treats digital psychiatry as a *programmable state space* rather than as a set of isolated forms. At its core, a multi-provider AI orchestration layer defines a mapping

$$\Phi_p : \Theta_{\text{canonical}} \longrightarrow \Theta_p$$

from a canonical sampling parameter space (temperature, top-p, maximum tokens, JSON constraints) to each provider-specific parameter space Θ_p. This abstraction allows model selection, sampling policies, safety constraints, and streaming behaviour to be specified once and instantiated consistently across commercial and local LLM backends. On top of this layer, a psychiatry feature module implements a section-driven architecture for encounters, a deterministic yet AI-ready MBC engine, and clinical summary templates with explicit provenance. Rating scales are represented as functions

$$f_{\text{MBC}} : \mathbf{x} \in \{0, \ldots, k\}^n \longmapsto \mathbf{z} \in \mathbb{R}^d,$$

where item responses $\mathbf{x}$ are mapped to both scalar scores and higher-dimensional feature vectors that can be consumed by AI models without sacrificing reproducibility of the underlying psychometrics.

Clinical flows in SynapseCore are formalised as finite-state machines

$$\mathcal{F} = (S, E, \delta, s_0),$$

where S is the set of clinical states (e.g., intake, risk formulation, medication review), E the set of events, and δ the transition function linking user actions

and MBC thresholds to next steps. A segment-aware session timer imposes a temporal index on these flows, yielding session traces of the form

$$\sigma = \left\{ (t_i, s_i, \mathbf{x}_i, \mathbf{z}_i, a_i) \right\}_{i=1}^{T},$$

where t_i denotes clock time, s_i the flow state, $\mathbf{x}_i$ raw inputs (narrative and scale responses), $\mathbf{z}_i$ derived features, and a_i AI calls and outputs. Tools and the Consulton workspace expose these structures as reusable "recipes" for common tasks such as intake synthesis, differential diagnosis scaffolding, crisis risk formulation, or longitudinal progress notes. An enhanced IDE-like surface permits informaticians and clinically interested developers to inspect and extend templates, calculators, and flow definitions within the same environment. Observability and telemetry components, together with a disciplined design-token system for layout and theming, provide a stable substrate for measuring latency, failure modes, and usage patterns while keeping computation and data client-side.

The main contributions of SynapseCore are fourfold. First, it provides a psychiatry-specific AI orchestration and configuration framework with *explicit, inspectable runtime state*, allowing clinicians and researchers to see which model was used, with which parameters, at which point in the clinical flow. Second, it embeds an MBC and knowledge layer that preserves deterministic scoring and published cut-offs, while simultaneously generating richer, vector-valued representations suitable for downstream analytics and AI assistance. Third, it couples flows and a session timer into a coherent temporal backbone, so that clinical segments, interventions, and AI outputs are linked into a single, reconstructable audit trail rather than scattered across documents and tools. Fourth, it presents these capabilities in an IDE-style, highly structured shell that makes digital psychiatry *programmable*: flows, templates, and scale configurations become editable artefacts rather than fixed properties of a closed system.

The evaluation strategy for SynapseCore is correspondingly multi-layered. Technical metrics (latency distributions, jitter, session timer drift, token efficiency) quantify whether the system behaves predictably under load. Content-level assessments involve expert review of generated text for clinical plausibility, factual reliability, and safety, with particular attention to risk-relevant segments such as suicidality or capacity formulations. Usability studies with practising clinicians examine whether the structuring of scales, flows, and AI prompts reduces subjective cognitive load and improves access to longitudinal information. Future work includes FHIR/EHR integration for real-world deployment, incorporation of more advanced predictive models for temporal trajectories and risk stratification, support for multi-user collaborative workflows (e.g., psychiatrist–psychologist–nurse teams), and adaptation of the framework to adjacent domains such as neurology, addiction services, and simulation-based training for psychiatry residents.

Keywords

Digital psychiatry; large language models; clinical decision support; measurement-based care; React/TypeScript; TensorFlow.js; telemetry; IDE-style clinical tooling.

Contents

Acknowledgements

The conceptual starting point for SynapseCore was not a single clinic or institution, but a recurring pattern in the literature on *digital psychiatry*, *measurement-based care*, and *electronic health records*: complex clinical work being conducted through fragmented tools, non-linear documentation, and interfaces that impose avoidable cognitive load. Reports of rating scales existing outside the main workflow, of progress notes drifting away from scale data, and of AI systems operating as opaque side-channels rather than integrated instruments were treated as constraints and failure modes to be designed against. SynapseCore is a concrete response to that pattern: a **browser-based workbench** that makes explicit how scales, flows, time, and AI outputs are linked at the level of the individual session.

Several design choices in SynapseCore are intentionally aligned with perspectives commonly described by people on the *autism spectrum*: a bias towards explicit structure, a preference for predictable state transitions, and low tolerance for hidden or implicit configuration. These preferences can be seen in the insistence on *visible flow graphs*, *deterministic scale calculators*, stable colour and typography systems, and an AI orchestration layer that exposes its parameters rather than burying them. The goal is not to aestheticise neurodiversity, but to take seriously the idea that clinical software should behave in a way that is legible to users who notice small inconsistencies, rely on routine, and are sensitive to noisy, ambiguous interfaces. If SynapseCore succeeds at all in this respect, it does so because such perspectives were treated as **design signals** rather than outliers.

This work is also indirectly shaped by a psychotherapeutic process that extended over approximately **twenty months**. That process was characterised by a disciplined repetition of small tasks, an explicit naming of cognitive and emotional patterns, and an emphasis on keeping difficult material within a bounded frame (a session, a set of notes, a concrete next step). SynapseCore encodes analogous principles at the software level: sessions are discretised and timed; flows enforce an explicit sequence of steps; measurement-based care is not an optional add-on but part of the default structure; AI output is attached to specific, inspectable states rather than appearing as free text detached from context. In this sense, the workbench functions as a *technical dedication* to that psychotherapy: a translation of the discipline and clarity of long-term therapeutic work into the logic of an application.

This dedication is also a recognition of the structural conditions under which many clinical psychiatrists currently work. They are expected to document complex trajectories, track longitudinal outcomes, and evaluate algorithmic tools, yet formal training in *computer science*, *software engineering*, *mathematics*, and *programming* remains limited or absent in most psychiatric curricula. As a re-

sult, documentation and follow-up practices often depend on digital infrastructures that were not designed with their cognitive workload, time constraints, or methodological needs in mind. SynapseCore is deliberately positioned in this gap: it attempts to embed **mathematical structure**, **software discipline**, and **code-level transparency** into the application itself, so that clinicians can benefit from these properties without being required to become programmers. The computational complexity is meant to reside in the system, not in the clinician's working memory.

The author's formal academic training lies *outside psychiatry*. This is stated plainly because the system is not intended to substitute for psychiatric education, supervision, or clinical responsibility. At the same time, digital psychiatry is inherently *transdisciplinary*. It depends on clinical science, psychometrics, software engineering, human–computer interaction, and increasingly on machine learning and AI safety. Within such a space, it is legitimate for contributors from other domains to propose tools, provided that the boundaries are explicit: **clinical judgement remains central**; regulatory and ethical frameworks are treated as non-negotiable constraints; and all algorithms and interfaces are designed to be inspectable and, when necessary, overridable by clinicians. The present work is written from that position: outside the specialty, but accountable to it.

In terms of potential contribution to the literature, SynapseCore is less a new algorithm than a synthesis of several strands that are usually discussed separately. Measurement-based care is implemented as a first-class, deterministic scoring engine; structured clinical flows are treated as finite-state machines that can be rendered, logged, and audited; a session timer acts as a *temporal backbone* that binds scales, narrative, and interventions into a single timeline; and AI models are orchestrated through a mapping layer that keeps sampling parameters, model identities, and prompts explicit. The entire system runs **client-side in the browser**, with an emphasis on local control and on minimising export of sensitive data. If there is a distinct methodological claim here, it is that this combination—**flows + scales + time + auditable AI**, held together in a local, inspectable workbench—constitutes a useful unit of design and analysis for future digital psychiatry tools.

SynapseCore would not exist without an extensive open-source and research ecosystem. The JavaScript and TypeScript toolchain, state-management and telemetry libraries, browser-based numerical and machine learning frameworks, and a large body of work on digital mental health, scale construction, and measurement-based care all serve as prerequisites for the present system. This manuscript should therefore be read as a small, structured artefact sitting on top of that foundation, attempting to be precise about how such components can be assembled into a workbench that is *technically constrained by privacy and transparency*, yet flexible enough to be adapted by clinicians and researchers in their own settings.

Finally, a minor practical remark: if SynapseCore occasionally appears *too* structured, this should be interpreted as a feature rather than a bug. It was designed for days when working memory is limited, interruptions are frequent, and the number of open browser tabs is already approaching a clinically relevant threshold. Any reduction in the density of sticky notes on a psychiatrist's monitor may be considered an unintended, low-intensity treatment effect of the system.

...to my lovely therapy sessions.

Chapter I: Concepts and Clinical Foundations

Sections 1–2

Overview. Part I introduces the clinical, technological, and design context for SynapseCore. Section 1 presents the background, problem statement, overview of the workbench, and contributions $\mathbf{C} = \{C_1, \ldots, C_6\}$, connecting them to the broader literature on multimorbidity, documentation burden, and large language models in psychiatry. Section 2 then details the target clinical scenarios, design principles, and the role of measurement-based care (MBC) as a core constraint on system design.

From a structural perspective, Part I defines the problem space and the high-level requirements that the architecture must satisfy. We write the requirement vector as

$$\mathbf{r} = (r_{\text{MBC}}, r_{\text{AI}}, r_{\text{workflow}}, r_{\text{observability}}, r_{\text{governance}}),$$

where each component captures a design obligation (for example, support for MBC, multi-provider AI, workflow integration, observability, and governance). The subsequent parts of the manuscript can then be interpreted as constructing an architecture that maps this requirement vector into a concrete implementation.

1 Introduction

1.1 Clinical and Technological Context

Contemporary psychiatric practice is characterised by escalating clinical and organisational complexity. Many patients present with multiple co-occurring psychiatric and somatic conditions across the life course, a pattern of multimorbidity that is associated with higher health-care utilisation, poorer functional outcomes, and more complicated treatment decision-making (Carlson & Yarns, 2023; Pati et al., 2021). Multimorbidity frequently coexists with extensive psychopharmacological regimens: population-based data show that older adults with severe mental disorders, such as schizophrenia, commonly receive both psychiatric and non-psychiatric polypharmacy, reflecting attempts to manage interacting symptom clusters, physical comorbidities, and iatrogenic side effects (Lunghi et al., 2023). Recent reviews emphasise that psychiatric polypharmacy is often driven by the heterogeneity, chronicity, and treatment refractoriness of mental disorders, but simultaneously raises concerns about adverse events, drug–drug interactions, and monitoring burden, thereby adding to clinicians' cognitive workload (Shekho, 2024). In parallel, psychiatrists are expected to integrate diverse psychotherapeutic modalities, psychoeducation, family interventions, and social prescribing within brief encounters and resource-constrained service configurations, further stretching the available cognitive and temporal bandwidth of clinicians.

Measurement-based care (MBC) has emerged as an evidence-based strategy to support clinical decision-making under these conditions. MBC denotes the routine, systematic use of validated symptom scales and patient-reported outcome measures to monitor progress and guide treatment adjustment over time (Kilbourne et al., 2018; Lewis et al., 2019). Randomised and quasi-experimental studies suggest that MBC is associated with better symptom outcomes, improved therapeutic alliance, and more structured treatment planning in outpatient psychiatry and behavioural health services (Kilbourne et al., 2018). At the same time, implementation research consistently shows that MBC remains underutilised in routine practice. Clinicians and staff frequently report that time burden, perceived workflow disruption, incomplete integration of measures with electronic health records (EHRs), and uncertainty about the actionable value of scores are major barriers to sustained adoption (Dey et al., 2025; Ko et al., 2023; Van Tiem et al., 2022). In telehealth and hybrid models of care, the logistical work of administering, scoring, visualising, and discussing measures can introduce additional friction if tools are not tightly integrated into the consultation workflow.

These clinical demands are superimposed on a fragmented digital environment. Psychiatrists typically navigate among EHR interfaces, standalone rating-scale platforms, external risk-calculator websites, guideline PDFs, and, increasingly, generic AI chat tools during or around the clinical encounter. Studies of EHR interaction design describe a phenomenon of "display fragmentation," in which clinically relevant information is spread across multiple screens and navigation paths, forcing clinicians to perform continuous mental integration of scattered data and increasing the risk of omission or error (Senathirajah et al., 2017, 2020). Narrative reviews and empirical studies link EHR usability problems to elevated cognitive load, documentation burden, and burnout, particularly when time spent on documentation, inbox management, and order entry substantially exceeds direct patient-facing time (Asgari et al., 2024; Holmgren et al., 2024; Tajirian et al., 2020). Analyses of documentation burden similarly highlight that many current measures of clinician workload are effectively proxies for EHR interaction time, reinforcing the perception of "desktop medicine" as a central stressor in mental health care (Alobayli, Alshammari, et al., 2023). Within this ecology, tools for MBC, risk assessment, and decision support are often deployed as additional, loosely coupled components rather than as first-class, programmable elements of an integrated workflow.

Against this backdrop, large language models (LLMs) have rapidly entered clinical discourse as potential "co-pilots" for documentation, summarisation, guideline retrieval, and decision support. Recent medical informatics reviews outline a broad landscape of LLM applications spanning clinical documentation, triage, question answering, and patient education, while emphasising persistent challenges related to hallucinations, bias, explainability, and regulatory oversight (Busch et al., 2025; Maity & Saikia, 2025a; Yu et al., 2025). In psychiatry specifically, systematic reviews report early use cases ranging from automated note drafting and diagnostic support to patient-facing conversational agents, relapse prevention tools, and educational resources, but stress that most systems remain research prototypes, evaluated in limited settings with heterogeneous outcome measures (Omar et al., 2024). Scoping work in digital mental health further suggests that generative models could support psychotherapy, safety planning, and longitudinal monitoring, yet calls for rigorous evaluation, robust safety frameworks, and careful integration into existing care pathways to avoid exacerbating inequities or promoting over-reliance on unverified outputs.

At the same time, bench-testing of LLMs in clinical decision-support scenarios indicates that hallucinations can arise frequently, particularly under adversarial prompts, ambiguous questions, or when models are asked to extrapolate beyond the explicit evidence present in their inputs (Kim et al., 2025; Omar et al., 2025). Such findings underscore that, although LLMs are powerful generative tools, naïve use through opaque, free-text prompts risks embedding unverifiable or non-reproducible reasoning into psychiatric workflows. Current implementations seldom connect model outputs to structured clinical constructs such as validated symptom scales, formal diagnostic criteria, or pre-defined clinical flows, limiting their suitability for MBC, quality improvement, and audit. Taken together, these trends delineate a context in which psychiatrists face high clinical complexity, documentation burden, and tool fragmentation, while LLM-based systems offer substantial promise but require careful, programmable integration with domain-specific workflows, safety constraints, and measurement infrastructures to realise their potential safely and reproducibly.

1.2 Problem Statement

The preceding section outlined how contemporary psychiatry is situated at the intersection of multimorbidity, polypharmacy, intensive documentation, and fragmented digital tools (Carlson & Yarns, 2023; Lunghi et al., 2023; Pati et al., 2021; Shekho, 2024). Measurement-based care (MBC) has been promoted as a way to structure decision-making under these conditions, yet implementation studies show that routine use of scales and patient-reported outcome measures remains limited, particularly when administration and interpretation are not seamlessly embedded into clinical workflows (Kilbourne et al., 2018; Ko et al., 2023; Lewis et al., 2019). In parallel, work on electronic health record (EHR) usability documents substantial display and task fragmentation: clinicians must traverse multiple screens and applications to assemble a coherent picture of the patient, which increases cognitive load and is associated with stress and burnout (Alobayli, Alshammari, et al., 2023; Asgari et al., 2024; Holmgren et al., 2024; Senathirajah et al., 2017, 2020; Tajirian et al., 2020). Taken together, these findings suggest that the central challenge is not merely the absence of individual tools (for scales, guidelines, or AI) but the lack of an integrated, programmable environment in which such tools can be orchestrated coherently.

Existing digital psychiatry solutions typically fall into one of three categories. First, commercial EHRs provide psychiatry-specific forms and order sets, but these are often rigid, difficult to customise, and not designed as programmable artefacts. Embedding new flows, scales, or calculators frequently requires vendor intervention, and local informatics teams have limited control over the underlying logic or user interface. Second, stand-alone web

applications and mobile tools provide specialised functionality (e.g., rating scales, risk calculators, or decision aids), but they operate outside the primary clinical workspace and rarely exchange structured data with the EHR or with each other. Third, a growing number of research prototypes and early products integrate large language models (LLMs) into psychiatric documentation or triage, typically by exposing a single-model chat interface that sits alongside existing systems rather than within a structured, domain-specific workflow (Busch et al., 2025; Maity & Saikia, 2025a; Omar et al., 2024; Yu et al., 2025). None of these categories, on their own, provide clinicians and local developers with a single, extensible workbench for designing and governing AI-enabled workflows.

Medical informatics reviews highlight both the breadth of LLM applications and the persistence of safety and governance challenges, including hallucinations, bias, and lack of transparency in how prompts and system messages are engineered (Busch et al., 2025; Maity & Saikia, 2025a; Yu et al., 2025). Bench-testing studies in clinical decision-support scenarios further show that LLMs are vulnerable to adversarial prompts and can generate confident but incorrect recommendations, raising the risk of embedding unverifiable reasoning into clinical documentation and decision-making (Kim et al., 2025; Omar et al., 2025). A key observation from this body of work is that LLM-based systems are often evaluated as monolithic "black boxes": prompts, sampling parameters, model selection, and context-window design are not represented as first-class, inspectable artefacts. For psychiatry, where nuanced formulation, longitudinal trajectories, and risk assessment are central, this lack of explicit structure and versioning is particularly problematic.

From the perspective of psychiatric practice and local informatics, these limitations translate into four concrete gaps. First, there is no single programmable workbench that encodes psychiatry-specific structure in a way that is both human-readable and machine-actionable: clinicians need a stable representation of encounters in terms of sections, cards, flows, and scales, rather than ad hoc text fields. Second, existing AI tools rarely support multi-provider orchestration. Institutions increasingly wish to combine commercial APIs (e.g., OpenAI, Anthropic, Gemini) with on-premise or open-source models (e.g., Ollama deployments) for reasons of cost, privacy, or capability, yet most clinical prototypes are tied to a single backend and do not expose a unified abstraction for model selection, configuration, and comparison (Busch et al., 2025; Yu et al., 2025). Third, clinical utilities such as structured flows, segment-aware timers, and consultation workspaces (for example, an internal "consultation notebook" that collates scales, prompts, and drafts) are rarely designed as tightly integrated components that can both consume and constrain AI outputs. Finally, the majority of available tools do not expose a developer-friendly environment akin to an integrated development environment (IDE) in which prompts, orchestration logic, and telemetry can be inspected, versioned, extended, and tested by local teams.

The problem addressed in this work is therefore the absence of an IDE-style, psychiatry-specific workbench that (a) represents clinical structure explicitly (sections, cards, flows, and scales); (b) orchestrates multiple AI providers through a single, configurable abstraction; (c) embeds clinical tools such as flows, timers, and consultation workspaces in a way that is natively coupled to AI capabilities; and (d) offers a developer-facing layer in which prompts, logic, and measurements can be iteratively refined. Without such a programmable environment, it is difficult to build AI-assisted psychiatric workflows that are transparent, reproducible, evaluable, and adaptable to local practice patterns and governance requirements.

1.3 Overview of the SynapseCore Workbench

SynapseCore is designed as a single-page, web-based workbench that treats digital psychiatry not as a collection of loosely coupled tools but as a programmable, IDE-style environment. The implementation is based on React and TypeScript, with a modular folder structure that separates AI orchestration, psychiatry-specific content, panel layout, IDE functionality, and observability. This design directly responds to the display and task fragmentation, documentation burden, and AI governance challenges described in the clinical informatics literature (Alobayli, Alshammari, et al., 2023; Asgari et al., 2024; Busch et al., 2025; Holmgren et al., 2024; Kim et al., 2025; Maity & Saikia, 2025a; Omar et al., 2024, 2025; Senathirajah et al., 2017, 2020; Tajirian et al., 2020; Yu et al., 2025). At a high level, SynapseCore exposes a clinician-facing interface that encodes psychiatry-specific structure while simultaneously acting as a programmable environment for informatics and engineering users.

1.3.1 SynapseCore AI Panel

The central entry point for AI-assisted interaction is the SynapseCore AI Panel, implemented in `SynapseCoreAIPanel.tsx`. Conceptually, this panel functions as a multi-provider orchestration surface rather than a monolithic "chat box." It binds together three concerns:

1. *Model orchestration and configuration.* The panel presents a unified configuration layer over multiple LLM backends (e.g., cloud APIs and on-premise deployments), exposing model selection, sampling parameters, safety toggles, and context-window options as explicit, inspectable settings rather than hidden implementation details. This addresses governance concerns highlighted in LLM reviews, where lack of visibility into prompts and parameters is cited as a barrier to safety and reproducibility (Busch et al., 2025; Maity & Saikia, 2025a; Omar et al., 2025; Yu et al., 2025).
2. *Conversation state and clinical context.* The AI Panel maintains a structured representation of the dialogue, including de-identification flags, encounter identifiers, and links to measurement-based care (MBC) artefacts, flows, and timer segments. Instead of treating text prompts as free-form input, the panel associates each exchange with explicit metadata (e.g., "risk formulation," "progress note draft," "psychoeducation"), facilitating downstream audit and evaluation.
3. *Bridging to the rest of the workbench.* Through typed props and shared state stores, `SynapseCoreAIPanel.tsx` receives signals from the Center Panel (e.g., current flow state, active timer segment, selected patient scales) and can push structured outputs back into psychiatry cards, flows, or IDE buffers. This bi-directional linkage is essential for embedding AI in clinical workflows rather than leaving it as a parallel, copy–paste-based activity.

1.3.2 Psychiatry Modal and Content System

The Psychiatry Modal, implemented in `PsychiatryModal.tsx`, provides a structured, domain-specific shell around encounters, sections, and content units. It organises information into a hierarchy of sections (e.g., "Presenting Concerns," "History," "Risk and Safety"), cards (discrete information units or tools), and content slices (atomic data elements such as scale results, summaries, or AI-generated narratives). This design is motivated by the need to represent psychiatric knowledge in a format that is both human-readable and machine-actionable.

Within this modal, content types are treated as first-class components: a "scale card" can

embed autoscore functions and longitudinal plots, a "narrative card" can host AI-generated text with provenance metadata, and a "risk card" can combine structured checklists with free-text formulation. The modal manages focus, keyboard navigation, and persistence, allowing clinicians to move between cards without losing context. At the same time, the content system exposes well-typed interfaces for the AI Panel and the IDE so that prompts can explicitly target particular sections or cards (for example, "summarise only the MBC scales and risk factors from the last three sessions") rather than operating on undifferentiated text. By encoding psychiatry-specific structure at the UI level, the modal provides a substrate for MBC, risk documentation, and quality improvement that goes beyond conventional note templates (Kilbourne et al., 2018; Lewis et al., 2019).

1.3.3 Center Panel Shell, Flows, Timer, and Tools

The Center Panel Shell, implemented in `CenterPanelShell.tsx`, acts as the main "canvas" for stateful clinical activities. It hosts three tightly integrated subsystems:

1. *Flows.* Flows represent structured clinical processes such as intake assessments, medication follow-ups, crisis visits, or longitudinal review sessions. Each flow is modelled as a stateful form with explicit steps, fields, and validation logic, and can be linked to corresponding sections and cards in the Psychiatry Modal. The shell manages which flow is active, how it is rendered, and how its state is exposed to the AI Panel for context-aware prompting (e.g., generating a first-draft note or explaining treatment options based on the fields already completed).
2. *Session timer.* The timer subsystem, accessed through the Center Panel, provides a segment-aware timing engine that can annotate the session into clinically relevant phases (e.g., "check-in," "assessment," "intervention," "documentation"). Laps and segments can be aligned with AI prompts and MBC events, enabling reconstruction of the temporal logic of the encounter and supporting audit and training use cases. This responds directly to concerns about invisible AI-mediated reasoning and the need for traceability in clinical decision support (Kim et al., 2025; Omar et al., 2025).
3. *Tools and Consulton workspace.* The Tools area (including the Consulton workspace) provides a flexible space for calculators, checklists, scale dashboards, and "scratchpad" prompts. Rather than scattering these utilities across separate applications, the Center Panel Shell treats them as embedded components that can be pinned, re-used, and composed into recipes (e.g., "PHQ-9 + GAD-7 + risk formulation template"). Outputs from tools can be handed off directly to the AI Panel for summarisation or documentation, avoiding manual copying and reducing the risk of transcription errors.

By concentrating flows, timing, and tools in a single shell, SynapseCore aims to counteract the display fragmentation and task switching that characterise many current EHR-based workflows (Alobayli, Alshammari, et al., 2023; Asgari et al., 2024; Holmgren et al., 2024; Senathirajah et al., 2017, 2020; Tajirian et al., 2020).

1.3.4 Enhanced IDE, File Explorer, and Apply-Plan System

SynapseCore also includes an Enhanced IDE surface, implemented in `EnhancedIDE.tsx`, that exposes the workbench itself as a programmable object. This environment is primarily aimed at informatics and engineering users but is intentionally co-located with the clinician interface. The IDE comprises:

- *A domain-aware file explorer*, which presents configuration files, prompt libraries, flow definitions, and calculator scripts as navigable artefacts rather than opaque back-end

configuration. Files can be opened, inspected, and modified using standard IDE affordances (tabs, diff views, search).

- *An AI-assisted apply-plan system*, in which LLMs help generate "plans" for modifications (for example, introducing a new flow or adding a domain-specific risk calculator) that are then applied to the local code or configuration using deterministic transformations. Crucially, the apply-plan system keeps AI-generated changes explicit, reviewable, and reversible, addressing safety concerns about unconstrained code generation in clinical software (Busch et al., 2025; Maity & Saikia, 2025a; Yu et al., 2025).
- *Tight integration with the clinical layer*, so that changes made in the IDE can be immediately reflected in test patients or sandbox modes within the Psychiatry Modal and Center Panel Shell. This supports rapid iteration on clinical content and workflows while preserving a clear separation between runtime data and configuration.

Treating configuration, prompts, and orchestration logic as first-class "code" entities within the IDE is a deliberate response to the observation that many LLM-enabled clinical tools are currently deployed as static, undocumented prompt-engineering artefacts rather than as governed software components (Omar et al., 2024, 2025).

1.3.5 Observability and Dual-Use Concept

Finally, SynapseCore incorporates observability features, including OpenTelemetry-based instrumentation (Hansen & Hasselbring, 2025; OpenTelemetry Authors, 2025), to capture structured traces, metrics, and logs across the AI Panel, Psychiatry Modal, Center Panel, and IDE. Instrumentation spans events such as model invocations, token usage, latency, timer events, and user interactions with flows and tools. These data allow teams to quantify performance characteristics (e.g., response times, failure rates), inspect how AI outputs are used in practice, and relate system behaviour to documentation burden and cognitive load concerns described in the EHR literature (Alobayli, Alshammari, et al., 2023; Asgari et al., 2024; Holmgren et al., 2024; Tajirian et al., 2020). Observability thus provides the empirical basis for evaluating whether the workbench mitigates or exacerbates the problems it is intended to address.

Across these components, SynapseCore is intentionally conceived as both a clinician-facing interface and a programmable environment. For clinicians, it offers a structured, psychiatry-specific surface that brings MBC, flows, timers, and AI assistance into a single, coherent workspace. For informatics and engineering users, it exposes the same environment as an extensible platform with clearly defined modules, configuration files, and telemetry, enabling local adaptation, rigorous evaluation, and governance of AI-enabled workflows in digital psychiatry.

1.4 Contributions

This work makes six interrelated contributions that together define SynapseCore as an IDE-style, psychiatry-specific workbench for programmable AI-assisted clinical workflows. Conceptually, these contributions span (i) a multi-provider AI orchestration engine with a normalised runtime configuration; (ii) a psychiatry knowledge framework that encodes sections, cards, and content slices as first-class objects; (iii) a reusable measurement-based care (MBC) engine; (iv) a flows framework for stateful, AI-enabled clinical processes; (v) a session timer and audit engine with optional learning of temporal patterns; and (vi) an integrated IDE and file system for AI-assisted refactoring of the workbench itself. Below, each contribution is described both at the software-architecture level and, where appropriate, with a compact mathematical formalisation.

C1: Multi-Provider AI Orchestration Engine with Normalised Runtime Config

The first contribution is a multi-provider AI orchestration engine that exposes a normalised runtime configuration and model registry across heterogeneous large language model (LLM) backends. This engine is implemented primarily in `src/ai/*` and coordinated at the state level by `src/stores/useAiConfigStore.*`. Instead of binding the clinical interface to a single AI provider, SynapseCore defines an abstraction over providers (e.g., OpenAI, Anthropic, Gemini, Ollama, or institutional custom models) that is expressed as a common configuration schema.

Formally, let

$$\mathcal{P} = \{\text{OpenAI}, \text{Anthropic}, \text{Gemini}, \text{Ollama}, \dots\}$$

denote the set of AI providers, and let each provider expose a finite set of models

$$\mathcal{M}_p = \{m_{p,1}, \dots, m_{p,k_p}\} \quad \text{for } p \in \mathcal{P}.$$

SynapseCore maintains a model registry

$$\mathcal{R} : \mathcal{P} \times \mathcal{M} \to \mathcal{C},$$

where $\mathcal{M} = \bigcup_{p \in \mathcal{P}} \mathcal{M}_p$ and $\mathcal{C}$ is a space of normalised configuration objects. Each configuration

$$c \in \mathcal{C}$$

is a tuple

$$c = (m, \theta, \sigma, \kappa, \mathbf{s}),$$

where m is a model identifier, θ encodes sampling parameters (e.g., temperature, top-p, maximum tokens), σ collects safety-related toggles and filters, κ describes context-window policies (e.g., truncation, summarisation strategies), and $\mathbf{s}$ contains provider-specific metadata mapped into a shared schema.

At runtime, a clinical task T (e.g., "draft intake summary", "explain risk formulation") is routed through a deterministic orchestration function

$$\Phi : (T, c, x) \mapsto y,$$

where x is the structured clinical context (e.g., selected flows, scales, and timer segments) and y is the generated output plus provenance metadata. The key property is that Φ is provider-agnostic: all provider-specific details are encapsulated in c and in adapter functions that conform to a common interface in `src/ai/*`. The store `useAiConfigStore` maintains the current configuration c_t and records transitions

$$c_{t+1} = f_{\text{update}}(c_t, u_t),$$

where u_t represents user or system actions (e.g., switching model, changing temperature, enabling de-identification). This normalised configuration and registry enable systematic comparison of models, reproducible auditing of AI behaviour, and explicit governance of which configurations are permissible in particular clinical contexts.

C2: Psychiatry Knowledge Framework Linking Sections, Cards, and Content Slices

The second contribution is a psychiatry knowledge framework that represents the clinical workspace as a graph of sections, cards, and content slices, each of which can be bound to AI prompts and structured forms. Within SynapseCore, encounters are not treated as flat

text; instead, `PsychiatryModal.tsx` and associated content components encode a hierarchical, typed model of the psychiatric consultation.

Let an encounter at time t be represented by

$$E_t = \big(S_1, \ldots, S_n\big),$$

where each section S_i is a tuple

$$S_i = \big(\text{name}_i, \mathcal{C}_i\big), \quad i = 1, \ldots, n,$$

and $\mathcal{C}_i$ is a set of card instances. Each card $c \in \mathcal{C}_i$ can be written as

$$c = \big(\text{type}(c), \text{schema}(c), \text{data}(c), \text{aiBindings}(c)\big),$$

where:

- $\text{type}(c)$ denotes a semantic category (e.g., "scale card", "risk card", "narrative card").
- $\text{schema}(c)$ describes the fields and constraints that govern the card's structured data.
- $\text{data}(c)$ contains the actual content (e.g., numeric scores, free-text formulation, AI-generated summaries).
- $\text{aiBindings}(c)$ specifies how the card participates in AI workflows (e.g., which prompts it contributes to, how its content is referenced and constrained).

This yields a labelled directed graph

$$G_t = (V_t, E_t^{\text{edges}}),$$

where V_t contains all section and card nodes at time t, and E_t^{edges} encodes relationships such as "card belongs to section", "card depends on scale", or "summary card aggregates from risk cards". AI prompts are constructed as functions over G_t:

$$\Psi : (G_t, \mathcal{Q}) \mapsto \mathcal{P},$$

where $\mathcal{Q}$ is a set of query specifications (e.g., "include only MBC and risk-related content", "exclude patient identifiers") and $\mathcal{P}$ is a set of prompt objects with explicit content provenance. This framework allows SynapseCore to generate context-aware prompts that are grounded in structured clinical entities rather than in loosely scoped text fragments, and it provides a natural hook for exporting structured representations (e.g., JSON, FHIR-like structures) for quality and research applications.

C3: Reusable Measurement-Based Care Engine with Autoscore and AI-Ready Reports

The third contribution is a reusable measurement-based care engine that provides autoscore functions and AI-ready reports for a broad range of psychiatric scales. The core implementation resides in `mbc/calculators.ts` and related modules. Instead of encoding scoring logic inside UI components, SynapseCore abstracts each scale as a pure function with well-defined input and output spaces.

Let $\mathcal{S}$ denote the set of supported scales (e.g., PHQ-9, GAD-7). For each scale $s \in \mathcal{S}$ with k_s items, the engine defines a scoring function

$$f_s : \mathbb{R}^{k_s} \to \mathbb{R}^d,$$

where the input is a vector of item responses (typically integer Likert scores) and the output $f_s(\mathbf{x})$ may include one or more scores: total severity, subscale scores, and categorical severity labels. For example, for a simple unidimensional scale,

$$f_s(\mathbf{x}) = \left(\sum_{j=1}^{k_s} w_j x_j,\ \ell\left(\sum_{j=1}^{k_s} w_j x_j \right) \right),$$

where w_j are item weights (often $w_j = 1$) and $\ell(\cdot)$ maps total scores into severity categories (e.g., minimal, mild, moderate).

Longitudinally, for a given patient and scale s, the engine maintains a time-ordered sequence of observations

$$\{(t_1, \mathbf{y}_1), \ldots, (t_T, \mathbf{y}_T)\}, \quad \mathbf{y}_i = f_s(\mathbf{x}_i),$$

and derives change metrics such as

$$\Delta \mathbf{y}_i = \mathbf{y}_i - \mathbf{y}_{i-1}, \quad i = 2, \ldots, T,$$

or standardised effect sizes

$$\mathrm{ES}_i = \frac{y_{i,\mathrm{total}} - y_{1,\mathrm{total}}}{\hat{\sigma}_{\mathrm{baseline}}},$$

where $y_{i,\mathrm{total}}$ denotes the total score component and $\hat{\sigma}_{\mathrm{baseline}}$ is an estimate of baseline variability across the clinic population.

On top of these numerical outputs, SynapseCore renders AI-ready HTML reports: structured documents that include tables of scores, sparkline-style trend plots, and machine-readable attributes (e.g., `data-scale="phq9"`). These reports are then passed to the AI orchestration layer as structured context, allowing models to generate narratives such as "symptom improvement", "partial response", or "worsening" grounded in actual numeric trajectories rather than in free-text notes. Because the scoring logic is centralised in `mbc/calculators.ts`, the same engine can be reused across the Psychiatry Modal, the Flows framework, and export pipelines, ensuring determinism and consistency in MBC outputs across the workbench.

C4: Flows Framework for Stateful Clinical Processes with AI Narrative Generation

The fourth contribution is a flows framework that encodes clinical processes as stateful forms with integrated AI narrative generation. Flows are implemented as structured components and configuration objects within the Center Panel, allowing clinics to represent common encounter types (e.g., intake, medication review, crisis visit, longitudinal review) as explicit process models.

Mathematically, a flow F can be represented as a labelled transition system

$$F = (X, \Sigma, \delta, x_0, X_F),$$

where:

- X is a finite set of states, each corresponding to a form step or sub-section.
- Σ is a set of events (e.g., "field updated", "step completed", "AI summary generated").
- $\delta : X \times \Sigma \to X$ is a transition function that determines the next state given the current state and an event.
- $x_0 \in X$ is the initial state.
- $X_F \subseteq X$ are terminal or "completeness" states (e.g., the flow is ready for signing or export).

Within each state $x \in X$, the flow defines a schema $\text{schema}(x)$ describing which fields must be captured, which MBC scales or cards are linked, and which AI prompts are relevant. As the clinician progresses through the flow, the system maintains a structured state

$$\mathbf{z}_t \in \mathcal{Z},$$

representing the partial encounter data at time t. AI narrative generation is then expressed as a mapping

$$\Gamma_x : (\mathbf{z}_t, c) \mapsto y_x,$$

where Γ_x is a prompt template associated with state x, c is the current AI configuration from the orchestration engine (C1), and y_x is a state-specific narrative (e.g., "presenting problems", "interim risk formulation", "treatment options discussed"). The narratives $\{y_x\}$ can be concatenated, transformed, or selectively included in the final note, with explicit links back to the structured fields that they summarise.

This formalisation allows flows to serve as the backbone of AI-assisted documentation: each step is both a set of fields and a narrative generation locus, and the transitions between steps are observable events in the telemetry layer. Clinics can thus design, test, and refine flows as programmable artefacts, rather than relying on static templates or ad hoc documentation patterns.

C5: Session Timer and Audit Engine with Optional TensorFlow.js-Based Pattern Learning

The fifth contribution is a session timer and audit engine that can optionally learn temporal patterns in session structure using TensorFlow.js. The core timing logic resides in `timerEngine.ts`, and learning hooks are implemented in `timerHooks/useSessionML.ts`. The timer is segment-aware: it does not merely record a scalar duration but instead tracks piecewise-constant segments with semantic labels (e.g., "check-in", "assessment", "intervention", "documentation").

Let a session be represented by a sequence of segments

$$\mathcal{T} = \{(s_1, \ell_1, \tau_1), \ldots, (s_K, \ell_K, \tau_K)\},$$

where s_k is a segment label from a finite set $\mathcal{L}$, ℓ_k is the duration of segment k, and τ_k is the start time. From this, SynapseCore derives a feature representation

$$\mathbf{x} = \phi(\mathcal{T}) \in \mathbb{R}^d,$$

for example by concatenating segment durations, normalised proportions, or simple summary statistics (e.g., ratio of assessment to intervention time). The optional machine learning component defines a model

$$g_{\boldsymbol{\phi}} : \mathbb{R}^d \to \mathbb{R}^q,$$

parametrised by weights $\boldsymbol{\phi}$, implemented in TensorFlow.js and trained on historical session data. Depending on configuration, $g_{\boldsymbol{\phi}}$ can be used to:

- cluster sessions into patterns (e.g., typical vs atypical structure),
- forecast remaining time for a given flow stage,
- or provide simple "session shape" descriptors that can be surfaced to clinicians (e.g., "today's session allocated more time than usual to safety planning").

Audit trails are built by combining the timer signal with AI events. Let

$$\{t_1^{\text{AI}}, \ldots, t_M^{\text{AI}}\}$$

denote timestamps at which AI calls occur during the session. For each call, SynapseCore records the active segment label $s(t_m^{\mathrm{AI}})$, the flow state, and the configuration c_{t_m} from C1. This yields an audit log

$$\mathcal{A} = \left\{ (t_m^{\mathrm{AI}}, s(t_m^{\mathrm{AI}}), c_{t_m}, T_m, y_m) \right\}_{m=1}^{M},$$

where T_m is the task (e.g., "draft assessment paragraph") and y_m is the AI output. Such logs are essential for retrospective analysis of AI usage patterns, debugging, and compliance with emerging governance requirements.

C6: Integrated IDE and File System for AI-Assisted Refactoring of the Workbench

The sixth contribution is an integrated IDE and file system that enables AI-assisted refactoring and extension of the workbench itself. The `EnhancedIDE.tsx` component and its associated modules expose the SynapseCore code and configuration space as a navigable, editable tree. Rather than treating prompts, flows, and calculators as hidden implementation details, the IDE surface treats them as first-class resources that can be inspected, versioned, and modified.

Let the workbench configuration state at time t be denoted by W_t, comprising:

- a set of prompt templates $\mathcal{P}_t$ (for the AI Panel and flows),
- a set of flow definitions $\mathcal{F}_t$,
- a set of MBC calculator specifications $\mathcal{S}_t$,
- and additional UI/layout configuration $\mathcal{U}_t$.

An "apply-plan" operation is then expressed as

$$W_{t+1} = A(W_t, p_t),$$

where p_t is a plan object generated with the help of an LLM (e.g., "add a new flow for brief medication follow-up using PHQ-9 and GAD-7", "refactor the risk card prompt to include the latest MBC scores") and A is a deterministic transformation function that applies the plan to the concrete files under version control. Crucially, p_t is not executed blindly: the plan is rendered as a human-readable diff proposal, and the clinician or informatics user can accept, modify, or reject it. This separates generative creativity (the LLM proposing changes) from authoritative control (the user applying changes), mitigating the risks associated with unconstrained AI-driven code generation in clinical systems.

The integrated file explorer and IDE thus turn SynapseCore into a self-hosted platform for continuous improvement: clinicians and informatics teams can co-design new flows, prompts, and calculators within the same environment in which they are used, while the underlying changes remain transparent, auditable, and revertible. In combination with the observability layer, this enables a full loop from identifying friction or failure modes (through telemetry) to proposing changes (through AI-assisted plans) to deploying and monitoring improvements as part of routine digital psychiatry practice.

Taken together, contributions C1–C6 define SynapseCore as a unified, psychiatry-specific workbench that treats AI, MBC, flows, timing, and configurability as tightly coupled components of a programmable clinical environment, rather than as isolated add-ons or opaque services.

1.5 Structure of the Manuscript

The remainder of this manuscript is organised into four conceptual parts that jointly develop SynapseCore from clinical motivation to formal models, implementation details, and evalua-

tion and governance. At a high level, we partition the numbered sections

$$\mathcal{S} = \{1, 2, \ldots, 19\}$$

into four disjoint subsets

$$\mathcal{S}_{\mathrm{I}} \cup \mathcal{S}_{\mathrm{II}} \cup \mathcal{S}_{\mathrm{III}} \cup \mathcal{S}_{\mathrm{IV}} = \mathcal{S}, \qquad \mathcal{S}_i \cap \mathcal{S}_j = \emptyset \text{ for } i \neq j,$$

corresponding to Parts I–IV as follows:

$$\mathcal{S}_{\mathrm{I}} = \{1, 2\}, \quad \mathcal{S}_{\mathrm{II}} = \{3, 4, \ldots, 12\}, \quad \mathcal{S}_{\mathrm{III}} = \{13, 14\}, \quad \mathcal{S}_{\mathrm{IV}} = \{15, 16, 17, 18, 19\}.$$

Intuitively, Part I develops the conceptual and clinical foundations; Part II presents the concrete architecture and implementation; Part III introduces formal models and learning components; and Part IV addresses evaluation, governance, and implications for practice.

Part I: Concepts and Clinical Foundations (Sections 1–2). Part I introduces the clinical, technological, and design context for SynapseCore. Section 1 presents the background, problem statement, overview of the workbench, and contributions (C1–C6), connecting them to the broader literature on multimorbidity, documentation burden, and large language models in psychiatry. Section 2 then details the target clinical scenarios, design principles, and the role of measurement-based care (MBC) as a core constraint on system design. From a structural perspective, Part I defines the "problem space" and the high-level requirements that the architecture must satisfy; we can think of it as specifying an initial requirement vector

$$\mathbf{r} = \left(r_{\mathrm{MBC}}, r_{\mathrm{AI}}, r_{\mathrm{workflow}}, r_{\mathrm{observability}}, r_{\mathrm{governance}}\right),$$

where each component captures a design obligation (e.g., support for MBC, multi-provider AI, workflow integration, observability, governance). Parts II–IV can then be interpreted as constructing, instantiating, and evaluating an architecture that maps this requirement vector into a concrete implementation.

Part II: Architecture and Implementation (Sections 3–12). Part II constitutes the core constructive contribution of the manuscript. It describes the SynapseCore workbench as a layered architecture that realises contributions C1–C6 in concrete modules and data structures.

- Section 3 provides a system-level overview: the technology stack, top-level module map, high-level data flow, and external integration points.
- Section 4 introduces the multi-provider AI orchestration engine (C1), including the runtime configuration, model registry, and provider-specific clients expressed through a normalised call interface.
- Section 5 details the psychiatry knowledge framework (C2), where sections, cards, and content slices form a typed graph that binds structured clinical data to AI prompts and forms.
- Section 6 formalises the MBC engine (C3) and its autoscore functions, defining scoring schemas, severity bands, and the generation of AI-ready HTML reports and export artefacts.
- Section 7 presents the flows framework (C4), treating clinical processes as stateful forms with explicit schema and transitions, and describing how AI narrative generation is attached to these states.
- Section 8 describes the session timer and audit engine (C5), including the timer core, UI, mapping of laps to clinical events, and hooks for session-level machine learning.

- Section 9 explains the clinical tools and Consulton module, showing how calculators, dashboards, and consultation workspaces are embedded in the Center Panel shell.
- Section 10 focuses on user interface, layout shells, and the Synapse theme system, covering layout primitives, typography, and theme tokens.
- Section 11 introduces the Enhanced IDE and developer tooling (C6), including the editor/AI bridges and use cases for local clinical informatics teams.
- Section 12 describes observability and telemetry, including OpenTelemetry integration, core metrics, AI route instrumentation, and the foundations for future analytics.

Formally, Part II can be viewed as constructing an architecture mapping

$$\mathcal{A} : \mathbf{r} \longmapsto \big(\mathcal{M}_{\text{AI}}, \mathcal{K}_{\text{psy}}, \mathcal{E}_{\text{MBC}}, \mathcal{F}_{\text{flows}}, \mathcal{T}_{\text{timer}}, \mathcal{D}_{\text{dev}}\big),$$

where $\mathcal{M}_{\text{AI}}$ is the orchestration layer, $\mathcal{K}_{\text{psy}}$ the psychiatry knowledge graph, $\mathcal{E}_{\text{MBC}}$ the MBC engine, $\mathcal{F}_{\text{flows}}$ the set of flows, $\mathcal{T}_{\text{timer}}$ the timer/audit subsystem, and $\mathcal{D}_{\text{dev}}$ the developer-facing IDE and tooling. Each of these artefacts is realised in concrete files and React/TypeScript modules, but the focus of Part II is on the system-level decomposition and the interfaces between components.

Part III: Formalisation and Learning Components (Sections 13–14). Part III provides a mathematical and algorithmic treatment of SynapseCore. Section 13 introduces formal models that capture the system as a typed graph, represent session timelines, and characterise flows as finite state machines. We define, for example, the clinical workspace at time t as a labelled graph

$$G_t = (V_t, E_t, \tau_t),$$

where V_t are nodes (sections, cards, timers, AI calls), E_t are edges (containment, dependency, temporal order), and τ_t is a typing function assigning each node and edge a semantic category. Section 14 then discusses session-level machine learning modules, including feature engineering for session patterns, TensorFlow.js model design, and data pipelines for pattern learning and prospective analytics.

From a global perspective, Part III formalises the transformation

$$(\mathcal{K}_{\text{psy}}, \mathcal{F}_{\text{flows}}, \mathcal{T}_{\text{timer}}) \longmapsto \big(G_t, g_{\boldsymbol{\phi}}\big),$$

where G_t provides a structured semantic view of the system state and $g_{\boldsymbol{\phi}}$ denotes the learned models over sessions and flows. This formal layer underpins future analytical and safety-oriented work while keeping the architecture in Part II grounded in explicit mathematical objects.

Part IV: Evaluation, Governance, and Implications (Sections 15–19). Part IV shifts the focus from construction to evaluation, governance, and implications for practice. Section 15 outlines the evaluation strategy, including technical benchmarks (latency, token use, timer drift), usability studies, clinical content validation, and safety/error analysis. Section 16 discusses ethical, legal, and regulatory considerations, including intended use, privacy and security, data governance, and alignment with emerging AI regulation in health care. Section 17 articulates methodological and practical limitations, while Section 18 surveys future work, such as EHR/FHIR integration, advanced risk models, team-based workflows, and domain generalisation. Section 19 closes the manuscript with a synthesis of the architecture and its implications for digital psychiatry.

If we denote by $\mathcal{C} = \{C_1, \ldots, C_6\}$ the set of contributions and by $\mathcal{E}$ the evaluation and governance space (benchmarks, studies, ethical constraints), Part IV can be viewed as studying a

mapping

$$\Xi : \mathcal{C} \longrightarrow \mathcal{E},$$

which associates each contribution with corresponding evaluation methods, evidence requirements, and governance considerations. The appendices then provide supplementary material—scale definitions and cut-off tables, selected code listings, system and flow diagrams, configuration and deployment notes, prompt templates, and telemetry specifications—that support reproducibility and implementation in other settings.

Overall, this four-part structure is intended to guide the reader from high-level concepts and clinical motivation, through concrete architecture and formal models, to evaluation and governance, while maintaining a consistent link back to the central goal: a programmable, psychiatry-specific workbench for safe and transparent AI-assisted clinical workflows.

2 Clinical and Design Foundations

2.1 Target Clinical Scenarios

SynapseCore is designed around a small number of high–impact clinical scenarios rather than isolated features. Let

$$\mathcal{X}_{\text{clin}} = \{\text{outpatient}, \text{CL/emergency}, \text{CAMHS}, \text{groups}\}$$

denote the current target scenario set. For each scenario $x \in \mathcal{X}_{\text{clin}}$, we model a structured requirement tuple

$$x = \big(P_x,\ \Omega_x,\ R_x,\ W_x,\ \Phi_x\big),$$

where P_x specifies the patient population (e.g., adults vs. children/adolescents), Ω_x the organisational setting (e.g., clinic, emergency department, liaison ward), R_x the dominant risk and legal frame (e.g., suicide risk, decisional capacity, safeguarding), W_x the workflow objects encoded in the workbench (flows, cards, timers, MBC reports), and Φ_x a set of measurement-based care (MBC) and documentation obligations. The concrete React/TypeScript modules in `src/centerpanel/*` and `src/features/psychiatry/*` can then be seen as realisations of $\{W_x\}_{x \in \mathcal{X}_{\text{clin}}}$ subject to the constraints in $\{R_x, \Phi_x\}$.

Outpatient consultations and follow-up visits. In adult outpatient psychiatry, clinicians are expected to perform comprehensive diagnostic formulations, risk assessments, and longitudinal treatment planning within time-limited sessions, often under MBC expectations and guideline-based standards for psychiatric evaluation (Kilbourne et al., 2018; Lewis et al., 2019; Silverman et al., 2015). Outpatient encounters typically involve iterative refinement of diagnoses, psychopharmacological regimens, and psychotherapeutic plans across multiple visits, combined with periodic review of symptom scales and functioning measures.

Within SynapseCore, this scenario corresponds to a relatively stable population $P_{\text{outpatient}}$ (adult patients under longitudinal care) and a low–to–moderate acute risk profile $R_{\text{outpatient}}$ (chronic suicidality, relapse prevention, medication adherence). The workflow set $W_{\text{outpatient}}$ is implemented primarily through:

- psychiatry content cards and seeds (`src/features/psychiatry/seeds/*.ts`) for formulations, psychotherapies, psychometrics, and case letters;
- MBC autoscore and reporting utilities (`mbc/calculators.ts`) that transform repeated scale data $\{s_t\}$ into structured summaries and AI-ready HTML reports;

- tthe Center Panel shell and timer (`src/centerpanel/CenterPanelShell.tsx`, `src/centerpanel/components/TimerModal.tsx`), which couple free-text documentation with time-stamped segments.

We can view the outpatient session at time t as a tuple

$$\Sigma_t^{\text{out}} = \big(H_t,\ M_t,\ S_t,\ N_t\big),$$

where H_t denotes history and mental state examination fields, M_t the current medication and side-effect profile, S_t the vector of scale scores, and N_t the AI-assisted narrative exported from the SynapseCore AI Panel. The architecture aims to ensure that each update $\Sigma_t^{\text{out}} \to \Sigma_{t+1}^{\text{out}}$ is expressed through explicitly typed operations (scale re-scoring, flow-state transitions, timer laps) rather than opaque free-text prompts, aligning the system with MBC and guideline-based practice (Kilbourne et al., 2018; Lewis et al., 2019).

Consultation–liaison psychiatry and emergency/capacity assessments. Consultation–liaison (CL) psychiatry and emergency department (ED) work present distinctive demands: high acuity, tight time constraints, multi-morbidity, and the need for rapid risk assessment and decisional capacity determinations in medically ill patients (Bourgeois et al., 2019; Health Service Executive, 2018; Kane et al., 2022; Ziukelis et al., 2023). Guidelines for initial psychiatric evaluation emphasise that even brief assessments should systematically cover key domains (history of present illness, risk, mental state, psychosocial context), with clear documentation of reasoning and recommendations (Silverman et al., 2015).

SynapseCore explicitly targets this scenario through the flows framework in `src/centerpanel/Flows/*`. Here, the scenario

$$x_{\text{CL}} = \big(P_{\text{CL}},\ \Omega_{\text{ED/ward}},\ R_{\text{acute}},\ W_{\text{flows}}, \Phi_{\text{legal}}\big)$$

is instantiated by a set of finite-state "flows" for acute risk and capacity:

$$\texttt{FlowId} \in \{\texttt{"safety"}, \texttt{"agitation"}, \texttt{"catatonia"}, \texttt{"bfcrs"}, \texttt{"lorazepam"}, \texttt{"capacity"}, \texttt{"observation"}, \texttt{"observationContainment"}, \texttt{"review"}\}.$$

defined in `flowTypes.ts` and parameterised via specific form states such as `CapacityFormState` or agitation/safety form types. For decisional capacity, `CapacityFormState` encodes the canonical four-abilities model (understanding, appreciation, reasoning, expression of a choice) drawn from Appelbaum and Grisso's framework (Appelbaum & Grisso, 1988; Fassassi et al., 2009). Formally, for a given consultation we model a capacity vector

$$\mathbf{c} = \big(c_{\text{U}}, c_{\text{A}}, c_{\text{R}}, c_{\text{C}}\big) \in \mathcal{L}^4,$$

where each component $c_\bullet$ takes values in a finite set of ordinal levels (e.g., `"adequate"`, `"partial"`, `"inaccurate"`, `"unable"`). The flow state and corresponding AI narrative are then constrained to be functions of $\mathbf{c}$ and the documented decision context d:

$$\text{Note}_{\text{capacity}} = f_\theta\big(d,\ \mathbf{c},\ \text{risk factors},\ \text{follow-up plan}\big),$$

with f_θ implemented as a parameterised prompt template plus model-specific completion, rather than free-form text. Similar formalisation applies to agitation, safety, and observation/containment flows, where the state space is defined over behavioural descriptors, interventions attempted, monitoring level, and de-escalation criteria. This encoding aligns with CL psychiatry standards and liaison service models of care, which emphasise transparent documentation of indication, least-restrictive alternatives, and step-down plans for high-risk interventions (Bourgeois et al., 2019; Health Service Executive, 2018).

Child and adolescent mental health (CAMHS) scenarios. Child and adolescent mental health services (CAMHS) operate under distinct legal, developmental, and multi-agency constraints. National service specifications and integrated care pathways highlight the need for age-appropriate consent/assent processes, safeguarding, school liaison, and family involvement across stepped-care models (Health and Social Care Board Northern Ireland, 2018; NHS England, 2018; Scottish Government, 2020; Whitmyre et al., 2024). In this scenario, the population P_{CAMHS} spans children and young people; the organisational setting Ω_{CAMHS} often includes community clinics, schools, and crisis services; and R_{CAMHS} incorporates safeguarding, developmental risk, and capacity/assent issues.

SynapseCore encodes this scenario via CAMHS-specific content modules and seeds:

- `src/features/psychiatry/content/camhsChildAdolescent.ts` defines `CamhsItem` structures containing clinical summaries, indications, contraindications, outcome measures, and print-ready HTML templates for, for example, "CAMHS intake & school liaison" reports.
- `src/features/psychiatry/seeds/camhs.ts` defines cards such as "Consent/Assent & Safeguarding Summary", with prompt variables for Gillick/Fraser competence, parental responsibility, information-sharing preferences, and safeguarding red flags.

At a formal level, CAMHS cards introduce additional state components related to legal guardianship and developmental stage. For a child or adolescent case, we can augment the clinical state vector to

$$\Sigma_t^{\text{CAMHS}} = \big(H_t,\ M_t,\ S_t,\ N_t,\ G_t,\ \Lambda_t\big),$$

where G_t captures caregiver/guardian configuration (who can consent to what) and Λ_t captures school context, multi-agency contacts, and safeguarding status. CAMHS-specific cards and AI prompts are constrained to be functions of $(H_t, M_t, S_t, G_t, \Lambda_t)$, ensuring that generated narratives explicitly reference consent/assent and safeguarding considerations rather than treating child and adolescent work as a simple subset of adult outpatient practice.

Group programmes and psychoeducation. Psychoeducational and skills-based group programmes are central to modern psychiatric care, particularly in psychosis, bipolar disorder, and common mental disorders. Multi-family psychoeducation and structured group interventions have demonstrated effects on relapse rates, symptom burden, and caregiver outcomes in both schizophrenia and bipolar disorder (Dixon et al., 2003; Fiorillo et al., 2015; Miklowitz et al., 2003). Service-level guidance emphasises standardised curricula, pre/post measures, and explicit safety and ground-rule frameworks.

In SynapseCore, the group-programme scenario x_{groups} is represented through group content items and seeds:

- `src/features/psychiatry/content/groupVisitsPrograms.ts` defines `GroupItem` objects with fields for clinical summary, indications, contraindications, outcome measures, and detailed HTML scaffolds for 6-session group psychoeducation programmes, including ground rules, session-by-session outlines, and measure slots.
- `src/features/psychiatry/seeds/groupsPrograms.ts` provides cards such as "Multi-Family Psychoeducation (Schizophrenia/Bipolar)" with variables for programme focus, facilitators, session frequency, and pre/post measures.

Let G denote a group programme with sessions indexed by $k = 1, \ldots, K$ (e.g., $K = 6$). For each session, we can define a minimal data tuple

$$\Gamma_k = \big(\text{topic}_k,\ \text{methods}_k,\ \text{homework}_k,\ \text{risk notes}_k,\ S_k^{\text{pre}},\ S_k^{\text{post}}\big),$$

where S_k^{pre} and S_k^{post} are vectors of pre- and post-session scale scores. The group templates and prompts in SynapseCore are designed so that a programme-level summary

$$\text{Summary}_G = g_\theta(\Gamma_1, \ldots, \Gamma_K)$$

is generated in a way that makes the use of evidence-based modules, risk management decisions, and MBC outcomes explicit, rather than relying on ad hoc free-text descriptions. This supports both individual-level care and service-level audit of group interventions.

Cross-scenario mapping and extensibility. The four scenarios above are not implemented as isolated silos; rather, they share a common representational backbone. We can define a mapping

$$\Psi : \mathcal{X}_{\text{clin}} \longrightarrow \mathcal{P}(\mathcal{W}),$$

where $\mathcal{W}$ is the global pool of workflow objects (flows, cards, timers, MBC calculators, AI prompts), and $\Psi(x)$ returns the subset of workflow objects activated for scenario x. In matrix form, we can conceptualise a binary incidence matrix

$$R_{x,w} = \begin{cases} 1, & \text{if workflow object } w \in \mathcal{W} \text{ is relevant in scenario } x, \\ 0, & \text{otherwise,} \end{cases}$$

so that each row corresponds to a scenario in $\mathcal{X}_{\text{clin}}$ and each column to a SynapseCore component (e.g., `"capacity"` flow, CAMHS consent card, group psychoeducation template, adult MBC report). Outpatient scenarios typically activate adult evaluation and MBC objects; CL/emergency scenarios activate acute-risk flows, capacity flows, and high-acuity timer/audit patterns; CAMHS scenarios activate child/adolescent consent, safeguarding, and school liaison cards; group scenarios activate group templates and repeated MBC measurement objects.

From a design perspective, this formalisation means that adding a new scenario (e.g., perinatal psychiatry, geriatric liaison, or specialty programmes) amounts to defining a new $x \in \mathcal{X}_{\text{clin}}$ and updating the incidence pattern $R_{x,\cdot}$ by linking the scenario to appropriate combinations of existing and new workflow objects. The underlying AI orchestration, timer engine, and IDE infrastructure remain unchanged, allowing informatics and engineering teams to extend the workbench to additional domains without re-architecting the core system.

2.2 Design Principles

The architecture of SynapseCore was guided by a small set of design principles $\{P_1, \ldots, P_6\}$ that translate the requirement vector $\mathbf{r} = (r_{\text{MBC}}, r_{\text{AI}}, r_{\text{workflow}}, r_{\text{observability}}, r_{\text{governance}})$ into concrete constraints on interaction design and software structure. Intuitively, these principles are intended to ensure that the workbench feels like a natural extension of clinical practice rather than a generic technical artefact. They also encode a stance on how generative AI should be integrated into psychiatry: not as an opaque "extra assistant" sitting beside existing systems, but as a programmable component whose behaviour is constrained by measurement-based care (MBC), risk frameworks, and local governance (Asgari et al., 2024; Busch et al., 2025; Kilbourne et al., 2018; Lewis et al., 2019; Omar et al., 2025; Senathirajah et al., 2017; Yu et al., 2025).

P1: Clinician-centric interaction with low friction. The first principle is that the workbench must minimise interaction cost for clinicians, particularly during time-pressured encounters.

We view interaction cost as a composite function

$$C_{\text{int}} = \alpha\, k + \beta\, s + \gamma\, m + \delta\, \ell,$$

where k denotes the number of clicks and keyboard actions required to complete a task, s the number of screen or context switches, m the number of modal/state transitions that interrupt the clinician's mental model, and ℓ the number of "lost" or duplicated documentation steps (e.g., copying content between systems). The coefficients $(\alpha, \beta, \gamma, \delta)$ encode relative weights that can vary across settings but are constrained by the evidence on EHR-related cognitive load and burnout (Alobayli, Alshammari, et al., 2023; Asgari et al., 2024; Holmgren et al., 2024; Senathirajah et al., 2017, 2020; Tajirian et al., 2020).

At the UI level, P1 yields three concrete rules: (i) high-frequency tasks (history, mental state, risk, and MBC review) must be reachable in one or two action steps from the main workspace; (ii) documentation, scales, flows, and AI narratives must coexist within a single visual frame (the Psychiatry Modal plus Center Panel) to avoid display fragmentation; and (iii) timer controls, flows, and AI interactions must support keyboard-first operation and predictable focus behaviour, so that clinicians can maintain eye contact with patients rather than repeatedly searching the screen. The section–card–flow architecture in `src/features/psychiatry/*` and the Center Panel shell are explicitly optimised for these constraints: for example, flows and cards are arranged so that the most common actions (recording risk, checking MBC, generating a brief summary) follow a single, stable path under a fixed visual hierarchy.

P2: Structured + generative hybrid. The second principle is that SynapseCore must combine rigid clinical structure with flexible generative capacity. We formalise an encounter at time t as a structured state Σ_t (sections, cards, flows, scales, timers) together with a generative narrative component N_t:

$$\Xi_t = \big(\Sigma_t, N_t\big).$$

The structured state Σ_t is defined over typed objects—forms, scale results, risk vectors, capacity judgements—each with explicit schemas and validation rules. In contrast, N_t is a set of AI-assisted narratives: intake summaries, risk formulations, progress notes, psychoeducation text, letters to referrers, and so on.

In design terms, P2 forbids "free-floating" generative content. Every AI call must be a function of well-defined structured inputs:

$$N_{t+1} = g_\theta\big(\Sigma_t, c_t, T_t\big),$$

where g_θ is the model- and prompt-specific generative map, c_t is the current AI configuration from the orchestration engine (Section 4), and T_t is an explicit task label (e.g., `"draft_risk_paragraph"`, `"summarise_mbc_trend"`). The psychiatry knowledge framework (Section 5) implements this principle by requiring that prompts are built from graph queries over sections, cards, flows, and MBC reports, rather than over arbitrary text. Clinically, this means that AI narratives remain anchored to observable data and recognised formulations, supporting MBC and guideline-based practice (Kilbourne et al., 2018; Lewis et al., 2019; Omar et al., 2024, 2025).

P3: Measurement-aligned and provenance-preserving AI. Given the centrality of MBC and risk documentation in psychiatry, P3 requires that all AI outputs that bear on assessment, formulation, or treatment planning be traceable back to specific structured inputs. We define a provenance mapping

$$\pi : N_t \to \mathcal{P}(\Sigma_t),$$

which associates each generative artefact (e.g., a risk paragraph) with the set of structured elements it actually used (scale scores, flows fields, timer segments). In the UI, this is surfaced through hover-over provenance tags, inline references to scales ("PHQ-9 total 18, severe range"), and exportable JSON that records which inputs contributed to each paragraph. P3 thus links the MBC engine, flows framework, and timer/audit subsystem in a way that allows clinicians and auditors to answer the question "which data did the model use to reach this phrasing?" rather than treating generative outputs as opaque text (Asgari et al., 2024; Kilbourne et al., 2018; Kim et al., 2025; Lewis et al., 2019; Omar et al., 2025).

P4: Programmability and local control. P4 states that psychiatry services must be able to treat the workbench as programmable infrastructure rather than as a fixed product. Concretely, flows, cards, prompts, and calculators are represented as configuration and code under version control, exposed through the Enhanced IDE and file explorer (Section 11). If we write W_t for the configuration state at time t, then local teams can apply change plans

$$W_{t+1} = A(W_t, p_t),$$

where p_t is a human-readable "plan" (often drafted with AI assistance) and A is a deterministic transformation. This separation between plan generation and authoritative application is designed to support governance and safety: changes to prompts, flows, or risk thresholds are reviewable, revertible, and documented, and can be aligned with local guidelines and legal requirements without depending on vendor releases (Busch et al., 2025; Maity & Saikia, 2025a; Yu et al., 2025).

P5: Observability, safe defaults, and graceful failure. P5 requires that the system be observable and fail in ways that are safe and legible to clinicians. Instrumentation via OpenTelemetry (Section 12) captures traces of AI calls, timer events, flow transitions, and export operations. From a design perspective, we assume that any component may degrade or become temporarily unavailable (e.g., an external AI provider), and we require that the remaining structured workflow continue to function. Formally, if we denote by f_{AI} the generative component of a given task and by f_{struct} the purely structured component, then the system must always admit a fallback

$$f_{\text{task}} = \begin{cases} f_{\text{struct}} + f_{\text{AI}}, & \text{if AI backends are available and authorised,} \\ f_{\text{struct}}, & \text{otherwise,} \end{cases}$$

with clear UI indicators of which branch is active. Clinicians can thus rely on flows, scales, and timers even when AI is degraded, and telemetry provides the data needed to monitor latency, failure modes, and token consumption (Asgari et al., 2024; Hansen & Hasselbring, 2025; Holmgren et al., 2024; OpenTelemetry Authors, 2025).

P6: Progressive disclosure of complexity and role-appropriate views. Finally, P6 acknowledges that SynapseCore serves multiple user personas—from busy clinicians and trainees to clinical informaticians and developers. The interface therefore implements progressive disclosure: core clinical views (Psychiatry Modal, Center Panel, MBC reports, session timer) present only the controls and information necessary for day-to-day practice, while advanced configuration (prompt templates, flow schemas, telemetry dashboards) is available behind explicit "IDE" and "admin" surfaces. Formally, we can model the visible state for a given role u as a projection

$$\text{view}_u(\Xi_t, W_t) = \Pi_u\big(\Xi_t, W_t\big),$$

where Π_u hides implementation detail that is irrelevant or potentially distracting for that role. This principle supports safety and usability: junior clinicians are not overwhelmed by configuration options, whereas senior clinicians, informatics teams, and researchers can access the full structure needed for local adaptation, evaluation, and research.

Together, P_1–P_6 operationalise the requirement vector $\mathbf{r}$ in a way that ties user experience, clinical epistemology, and software architecture into a single design space. Subsequent sections describe how these principles are instantiated in concrete modules (Parts II and III) and how they inform the evaluation and governance agenda (Part IV).

2.3 Measurement-Based Care as a Core Constraint

Measurement-based care (MBC) is treated in SynapseCore not as a "nice-to-have" feature but as a hard constraint that shapes how data are collected, stored, and surfaced to the clinician. In line with contemporary guidelines on routine outcome monitoring and stepped-care models in psychiatry, MBC is defined here as the systematic use of validated rating scales at baseline and follow-up, with explicit rules for how changes in scores inform treatment decisions, risk assessment, and psychoeducation. In practical terms, this translates into three non-negotiable properties for every supported scale: (i) deterministic computation, (ii) traceability in code, and (iii) lossless convertibility into human-readable summaries and AI prompts.

Core instruments and first-wave implementation. In the current implementation, the MBC engine centres on a small but clinically high-yield set of instruments: the Patient Health Questionnaire–9 (PHQ-9) for depressive symptoms, the Generalized Anxiety Disorder–7 (GAD-7) for anxiety, the PTSD Checklist for DSM-5 (PCL-5) for trauma-related symptoms, together with the Yale–Brown Obsessive–Compulsive Scale (Y-BOCS) and the AUDIT-C alcohol screen. Each of these measures is represented by an explicit scoring function in `mbc/calculators.ts` (for example, `phq9Score`, `gad7Score`, `pcl5Score`) and is wired into an autoscore dispatcher via the discriminated union

$$\texttt{MeasureId} \in \{\texttt{'phq9'}, \texttt{'gad7'}, \texttt{'pcl5'}, \texttt{'ybocs'}, \texttt{'auditc'}\}.$$

Legacy content in `psychiatry/legacy/mbc.ts` exposes these same measures as part of a "Measurement-Based Care Overview" card, which combines serial PHQ-9 / GAD-7 scores with simple response/remission rules and adherence/side-effect anchors as a scaffold for less experienced clinicians.

Deterministic scoring and banding. Formally, each measure m is defined by a scoring function

$$f_m : \mathbb{Z}^n \to \texttt{ScoreResult},$$

where $\mathbb{Z}^n$ denotes the vector of item responses (e.g., 9 items for PHQ-9, 7 for GAD-7, 20 for PCL-5) and `ScoreResult` is the canonical output type:

```
export type ScoreBand = { label: string; min: number; max: number };
export type ScoreResult = {
  total: number;
  severity: string;
  bands: ScoreBand[];
  flags: string[];
  breakdown?: Record<string, unknown>;
};
```

To guarantee determinism, raw responses never flow directly into totals. Instead, a shared helper `coerce(items, len, lo, hi)` enforces three invariants:

1. *Fixed length:* arrays are truncated or padded with zeros to the instrument's item count n.
2. *Valid range:* all values are clamped to the instrument's allowed ordinal range $[lo, hi]$ (for example, 0–3 for PHQ-9/GAD-7, 0–4 for PCL-5 and Y-BOCS).
3. *Numeric safety:* non-finite or missing entries are converted to 0 before summation.

The core total is then computed by a simple reduction,

$$\texttt{total} = \sum_{i=1}^{n} x_i,$$

implemented via a shared `sum` helper. Severity is derived not by branching on arbitrary thresholds scattered throughout the UI, but by selecting from an explicit list of bands:

$$\texttt{bands} = \big[(\ell_1, a_1, b_1), \ldots, (\ell_k, a_k, b_k)\big],$$

where each triad (ℓ_j, a_j, b_j) corresponds to a level label, minimum, and maximum (for example, PHQ-9 bands "None" 0–4, "Mild" 5–9, "Moderate" 10–14, "Moderately severe" 15–19, "Severe" 20–27). The reported `severity` is obtained by a pure lookup

$$\texttt{severity} = \texttt{band.label} \text{ such that } \texttt{total} \in [\texttt{band.min}, \texttt{band.max}],$$

or "Unknown" if no interval matches. This design ensures that any change in cut-offs or band labels can be reviewed and versioned in a single file, rather than duplicated across forms, prompts, and exports.

Clinically meaningful flags and breakdowns. Beyond a scalar total, several measures incorporate clinically meaningful decision points as structured flags. For PHQ-9, item 9 (suicidal ideation) is treated specially; if the coerced value for item 9 exceeds 0, the scoring function appends a flag such as

```
'Item 9 > 0 -- discuss safety today'
```

to the `flags` array, ensuring that any downstream consumer (UI, AI, or export) can highlight the need for a same-day risk conversation. For PCL-5, the scoring function replicates DSM-5 cluster logic: symptoms are grouped into clusters B (intrusions), C (avoidance), D (negative alterations), and E (arousal), and an internal helper verifies that each cluster satisfies the usual "number of items rated ≥ 2" criterion. The result is returned as a `breakdown` object (for example, `{B: true, C: true, D: false, E: true}`), and additional flags such as

```
'Total ≥ 33 -- positive screen'  or  'Cluster criteria met (B≥1, C≥1, D≥2, E≥2)'
```

are added when relevant. In this way, the engine does not stop at a total score but encodes the intermediate clinical reasoning steps in a machine-readable form.

Traceability and single source of truth. Every scoring rule lives in `mbc/calculators.ts` as a standalone pure function with explicit types. The rest of the system—React components, legacy cards, prompt templates, and export routines—refer only to the exported interface (`ScoreResult`, `MeasureId`, and the `renderAutoscoreHTML` helper) and never implement their own arithmetic. This yields several traceability guarantees:

- A discrepancy between a displayed score and the underlying items can always be traced back to a single code path.
- Unit tests can be written directly against the scoring functions (for example, regression tests for known PHQ-9/GAD-7 scenarios) without involving the UI.
- Version control provides a complete audit trail of changes in cut-offs, flag wording, and severity labels, which is essential when aligning with institutional or national guidelines.

The legacy MBC card definitions in `psychiatry/legacy/mbc.ts` respect this constraint by restricting themselves to HTML layout, form scaffolding, and prompt templates. Whenever they need scores or classifications, they consume values produced by the calculators layer rather than recomputing totals internally.

From raw answers to human narratives and AI prompts. To support documentation and AI augmentation, the MBC engine exposes an autoscore renderer

```
renderAutoscoreHTML(measure, answers, opts)
```

that accepts a `MeasureId`, a raw response vector, and optional metadata (title, patient label, date, sex). Internally, this helper:

1. Normalizes responses with `coerce` and dispatches to the appropriate scoring function (`phq9Score`, `gad7Score`, `pcl5Score`, `ybocsScore`, or `auditCScore`).
2. Constructs a compact HTML "mini-page" reporting the total, severity band, any flags, and a table of item-level responses.
3. Generates an "anchors" string that serializes the severity bands (for example, "Mild: 5–9 ůů Moderate: 10–14 ůů Severe: 20–27") to support quick visual scanning and later use in AI prompts.

The resulting HTML can be embedded directly into the SynapseCore centre panel (for example, in a right-hand "MBC summary" rail) or converted to Markdown/plain text for insertion into clinical notes. Prompt templates in the MBC legacy cards then treat scale outputs as first-class variables:

```
Scales today -- PHQ-9: {{phq9}}; GAD-7: {{gad7}};
Functioning: {{func}}/10; Side-effects: {{se}}/3;
Adherence: {{adh}}%. Impression: {{impression}}. Plan: {{plan}}.
```

This pattern ensures that large language models never infer or re-compute scores; instead, they receive authoritative numerical values and flags as structured context and are asked only to help with linguistic formulation (for example, drafting a brief progress note paragraph consistent with the scores).

Form examples and clinical workflow. On the UI side, the "Measurement-Based Care Overview" card in `psychiatry/legacy/mbc.ts` demonstrates how MBC is woven into everyday encounters. The card is divided into three logical segments:

1. **Outcomes block (A):** a grid capturing baseline and current PHQ-9 and GAD-7, alongside simple sliders for functioning (0–10), side-effects (0–3), and adherence (0–100%). These values can be autoscore-linked or entered from external systems.
2. **Response classification (B):** a section that operationalises treatment response categories (for example, "Response" $\geq$50% reduction from baseline, "Partial response" 25–49%, "Non-response" <25% or worsening), together with remission criteria (for example,

PHQ-9 $\leq$4 sustained over $\geq$2 visits). Computed percentage change fields (such as `phq9_delta`) are intended either for automatic calculation via the MBC engine or manual entry in less integrated settings.

3. **Plan and communication (C):** free-text fields for the clinician's impression, shared decision-making notes, and planned next steps, with embedded prompt buttons that can call the AI orchestration layer to draft, refine, or translate the narrative while preserving the numerical content.

This structure reflects a deliberate design choice: scales are always shown in relation to time (baseline vs current), functioning, and side-effects, discouraging the common mistake of treating a single PHQ-9 score as a standalone diagnostic statement. The card acts as a "visual proof" that MBC in SynapseCore is about trajectories and decisions, not only checklists.

Schemas as constraints on future extensions. By insisting that every instrument conforms to the same `ScoreResult` contract and passes through the autoscore renderer, SynapseCore can safely extend the MBC portfolio (for example, adding insomnia, panic, or child/adolescent scales) without redesigning workflows. Any new measure must:

1. Declare its item count and allowed response range.
2. Implement a pure scoring function that returns `ScoreResult`, including any clinically relevant flags and breakdowns.
3. Register a `MeasureId` and, if needed, customize rendered labels via `renderAutoscoreHTML`.
4. Integrate into the cards/prompt layer only through these typed interfaces.

In summary, measurement-based care in SynapseCore is encoded as a structural property of the system: scales are first-class, their computations are centralized and verifiable, and their outputs are always available both as rigorous numeric objects and as ingredients for human-facing narratives and AI-assisted documentation.

2.4 User Personas and Workflows

User-centred design work in clinical systems repeatedly shows that explicit personas and end–to–end workflow maps are essential for safe, adoptable decision-support tools. In SynapseCore we treat these personas and workflows as first-class design constraints: they inform the data model, the orchestration between modules, and even the shape of the exported session artefacts.

We model a persona P abstractly as

$$P = (\text{name}, \text{role}, \text{goals}, \text{constraints}, \text{primary\_modules}),$$

and we insist that any new feature or refactor can be traced back to a concrete improvement in at least one P. In the current implementation there are three canonical personas:

- Persona 1: front-line clinician using the psychiatry shell, AI psychiatry panel and the session timer as a single clinical surface.
- Persona 2: clinical informatics expert or technically inclined psychiatrist using the Enhanced IDE and AI tools to extend, audit and parameterise the workbench.
- Persona 3: researcher or quality-improvement lead exploring longitudinal measurement-based care (MBC) data and session patterns via logs and exports.

The remainder of this subsection makes these personas concrete and links each of them to specific modules in the codebase and to a formal workflow schema.

Persona 1: Clinician using psychiatry shell + AI panel + timer. Persona 1 is an outpatient psychiatrist (or clinical psychologist) seeing high-volume patients in 30–60 minute sessions. Their goals are: (i) to run a safe, empathic interview; (ii) to capture MBC scores and risk-relevant information without breaking conversational flow; and (iii) to leave the session with a defensible note and a minimal amount of after-hours typing.

In the current codebase, Persona 1 interacts primarily with:

- the psychiatry feature layer in `src/features/psychiatry/` (content, section hierarchy, MBC cards, registry);
- the therapy timer stack in `src/centerpanel/components/TimerModal.tsx`, backed by the pure engine in `timerEngine.ts` and the audit/export logic in `state/timerAudit.ts`;
- the psychiatry AI content and scoring examples in `src/components/ai/psychiatry/` (`prompts.ts`, `forms/htmlForms.ts`, `score.ts`);
- the SynapseCore AI panel in `src/components/ai/panel/SynapseCoreAIPanel.tsx`, which provides the general-purpose coding/chat interface but also psychiatry-aware modes.

Formally, one outpatient visit can be represented as a sequence of three phases:

$$\text{Visit} = (\text{pre}, \text{in}, \text{post}),$$

with each phase mapped to explicit user actions and state transitions.

Pre-session preparation. In the pre-session phase the clinician loads the patient context and MBC scores.In practice this means:

- entering or confirming PHQ-9/GAD-7/PCL-5 scores via the MBC cards defined in `src/features/psychiatry/legacy/mbc.ts` and the calculators in `src/features/psychiatry/mbc/calculators.ts`;
- optionally using the HTML forms for PHQ-9/GAD-7 in `src/components/ai/psychiatry/forms/htmlForms.ts` to embed scale capture into an AI-assisted note or export;
- opening the Timer Modal (`src/centerpanel/components/TimerModal.tsx`) from the centre panel and selecting an initial segment kind (assessment/therapy/documentation) if desired.

At this point the core timer state is an instance of `Engine.TimerState`:

```
// src/centerpanel/components/timerEngine.ts
export interface TimerState {
  mode: Mode;              // "stopwatch" | "countdown"
  phase: Phase;            // "idle" | "running" | "paused" | "finished"
  elapsedMs: number;
  countdownInitialMs: number;
  remainingMs: number;
  laps: Lap[];
  segments: Segment[];
  zeroBehavior: ZeroBehavior;
  autoStopAtZero: boolean;
  _idCounter: number;
}
```

We can view the pre-session clinical state as

$$S_{\text{pre}} = (\text{patient\_id}, \text{encounter\_id}, \mathbf{m}_0, t_0),$$

where $\mathbf{m}_0$ is the baseline MBC vector (e.g. PHQ-9, GAD-7, PCL-5, functioning), and t_0 is the initial `TimerState` snapshot.

In-session flow. During the live session the Timer engine, psychiatry content, and AI panel operate as a coordinated triad:

- The session timer is driven by `useTimerEngine.tsx`, which wraps `Engine.start`, `pause`, `lap`, and `startSegment` with a React store and optional audit hooks in `state/timerAudit.ts`.
- Segments (`SegmentKind = "assessment" | "therapy" | "break" | "documentation"`) and laps are used by the clinician to mark clinically salient points: start/end of MSE, risk discussion, intervention modules, and documentation time.
- Concurrently, the psychiatry AI prompts (`prompts.ts`) provide structured scaffolds for MSE, risk triage, and formulation, which can be called into the AI panel either directly (via the psychiatry AI shim in `components/ai/psychiatry/`) or indirectly via the general `SynapseCoreAIPanel`.
- MBC scores can be re-collected mid-session (e.g. after a brief intervention block), using the same calculators in `mbc/calculators.ts`, preserving determinism and traceability.

If we denote by τ the full timer trajectory over a session, and by $\mathbf{m}_1$ the updated MBC vector at the end, then Persona 1's in-session workflow can be abstracted as

$$S_{\text{pre}} \xrightarrow{\text{start + segments + laps}} (S_{\text{in}}, \tau) \xrightarrow{\text{update MBC}} S'_{\text{in}},$$

with τ fully reconstructible from the stored `TimerState` and the audit functions in `timerAudit.ts`.

Post-session documentation. At the end of the visit the clinician needs a concise, defensible clinical artefact. SynapseCore supports at least three outputs aligned with Persona 1:

1. A timer audit report produced by `timerAudit.htmlReport(rec)`, where `rec` is a `TimerAuditRecord` summarising duration, segment percentages, and lap splits. This yields a human-readable HTML summary (with explicit disclaimer) while preserving the machine-parseable snapshot.
2. An MBC summary, where the scale results $\mathbf{m}_0, \mathbf{m}_1$ are represented as `ScoreResult` instances (total, severity band, flags) and combined with clinically meaningful deltas (e.g. response vs. remission).
3. A narrative note, generated in the AI panel using psychiatry-specific prompts (`MSE`, `RISK_TRIAGE`) and a structured MBC snippet template (e.g. the "Progress Note Snippet" template in `legacy/mbc.ts`), which remains editable text in the note store.

This can be viewed as a mapping from session state to documentation:

$$f_{\text{doc}} : (S'_{\text{in}}, \tau) \mapsto O_{\text{clin}},$$

where O_{clin} is a bundle consisting of (i) a narrative note, (ii) an MBC block, and (iii) an optional timer audit export. The important constraint is that each component is derived from deterministic, inspectable code paths (`calculators.ts`, `timerEngine.ts`, `timerAudit.ts`) rather than from opaque AI heuristics.

Persona 2: Clinical informatics expert using the Enhanced IDE. Persona 2 is a clinician–developer or informatics specialist responsible for maintaining and extending SynapseCore in a particular setting (e.g. a hospital department, a group practice, or a residency training programme). Their primary goals are: (i) to ensure that scoring rules, prompts, and exports

implement local policy and evidence-based practice; (ii) to preserve determinism and cross-language parity for critical computations; and (iii) to integrate SynapseCore with adjacent systems (EHRs, research pipelines, policy templates).

The main touchpoints for Persona 2 are:

- the Enhanced IDE in `src/components/ide/EnhancedIDE.tsx`, which wraps the Monaco editor, file explorer, AI assistant, and SynapseCore AI panel into a single developer surface;
- the AI-assisted development tools (`AiAssistant`, `SynapseCoreAIPanel`) wired into the IDE header and command palette;
- the MBC and psychiatry-specific modules in `src/features/psychiatry/mbc/`, `src/features/psychiatry/legacy/`, and `src/components/ai/psychiatry/`;
- the observability layer in `src/observability/` and the AI telemetry events in `src/components/ai/telemetry/events.ts`.

We can think of Persona 2 as operating in a "meta-workflow" that transforms the codebase C under a set of clinical invariants Φ (e.g. "PHQ-9 scoring must match published cutoffs; timer audit reports must not contain PHI when exported"). A single development cycle can be sketched as:

$$(C, \Phi) \xrightarrow{\text{edit in IDE + AI assist}} C' \xrightarrow{\text{type-check + tests}} C'' \xrightarrow{\text{deployment}} \text{Workbench}^*,$$

where Workbench* is the updated runtime available to Persona 1 and Persona 3.

Concretely, Persona 2 uses:

- the IDE's file explorer and editor to locate and edit scoring functions such as `phq9Score`, `gad7Score`, and `pcl5Score` in `mbc/calculators.ts`, referencing example implementations in `components/ai/psychiatry/score.ts`;
- the AI panel in "implement" mode (configured in `SynapseCoreAIPanel.tsx`) to generate refactor proposals or cross-language ports, while keeping the final TypeScript and Python examples aligned;
- telemetry helpers such as `withSpan` in `observability/spans.ts` and `emitAiRouteChanged` in `observability/aiRouteTelemetry.ts` to instrument new flows and evaluate their impact on latency and error rates;
- the shared type shapes (e.g. `ScoreResult`, `TimerAuditRecord`) as contracts ensuring that new UI fragments, exports, or tools remain compatible with existing data pipelines.

In this sense, Persona 2 enforces a form of contract-based design: any new workflow element for Persona 1 (e.g. an added scale, a new export template) must correspond to an explicit schema in the code, covered by tests or at least by type-level guarantees, and verifiably free of non-deterministic behaviour in core clinical computations.

Persona 3: Researcher exploring MBC and session patterns via logs and exports. Persona 3 is a researcher, quality-improvement lead, or resident engaged in systematic analysis of outcomes, adherence, and efficiency. Their focus is not the live encounter but the aggregated traces of many encounters: distributions of PHQ-9 change scores, time allocation between assessment and therapy, or associations between timer patterns and remission.

From the perspective of the codebase, Persona 3 is primarily a consumer of:

- the MBC engine outputs (`ScoreResult` objects for PHQ-9, GAD-7, PCL-5, and other scales);
- the timer audit exports via `TimerAuditRecord` and `htmlReport`, which encode segment durations, lap splits, and session snapshots;

- the AI streaming snapshots managed in `src/services/stream/snapshots.ts`, which store approximate token counts and last responses for each conversation;
- the session persistence layer in `src/centerpanel/SessionPersistence.tsx`, which serialises note state, MBC values, flows, and meta-data into a single blob;
- the optional TensorFlow.js session pattern model in `src/centerpanel/timerHooks/useSessionML.ts`, which exposes an interface for local pattern learning and prediction.

A single exported session can be viewed as a record

$$r = (\mathbf{x}, \mathbf{m}, \mathbf{d}, \ell, \mathbf{s}),$$

where:

- $\mathbf{x}$ encodes contextual features (time of day, day of week, clinician identifier, visit type), often derived from the `SessionPattern` interface in `useSessionML.ts`;
- $\mathbf{m}$ contains MBC outputs (baseline and current `ScoreResult` objects for each scale, plus derived flags such as response/remission);
- $\mathbf{d}$ summarises timer-derived durations, e.g. the vector $(d_{\text{assessment}}, d_{\text{therapy}}, d_{\text{break}}, d_{\text{documentation}})$ and the corresponding percentages computed in `timerAudit.ts`;
- ℓ holds lap-based microstructure (number of laps, mean split duration $\bar{d}_{\text{split}}$, labels at clinically significant points);
- $\mathbf{s}$ contains AI usage summaries (e.g. number of AI calls, token counts per route, note templates used), approximated from `SnapshotManager` snapshots and AI telemetry events.

This induces a dataset

$$D = \{r_i\}_{i=1}^{N},$$

which can be exported as HTML, CSV, or JSON artefacts depending on the surrounding infrastructure. At the code level, a minimal session export schema might be expressed as:

```
// conceptual shape for research exports
export interface SessionExportRecord {
  sessionId: string;
  patientPseudoId: string;   // de-identified
  startedAt: number;
  endedAt: number;
  mbc: {
    phq9?: ScoreResult;
    gad7?: ScoreResult;
    pcl5?: ScoreResult;
    // other scales...
  };
  timer: TimerAuditRecord;
  ai: {
    totalPrompts: number;
    totalTokensApprox: number;
    lastSnapshot?: Snap;
  };
}
```

Persona 3's workflow, unlike Persona 1's, happens largely outside the running application: exported `SessionExportRecord` objects are ingested into R, Python, or statistical packages, and linked to outcome labels or covariates. The tight constraint from SynapseCore's perspective is that the exported fields are:

1. traceable to specific pure functions in the codebase (e.g. `phq9Score`, `segmentTotals`);
2. de-identified by construction (e.g. using pseudo-identifiers rather than raw MRNs);
3. stable enough to support re-analysis over time, even as UI layers evolve.

Cross-persona invariants. Finally, these three personas are not independent silos but different projections of the same underlying system. In practice, SynapseCore enforces a small set of invariants that must hold simultaneously for Persona 1 (clinician), Persona 2 (informatics expert), and Persona 3 (researcher):

1. **Deterministic core logic.** All clinically relevant computations (MBC scores, timer segment totals, risk flags) must be implemented as pure, testable functions with explicit types (e.g. `phq9Score`, `gad7Score`, `pcl5Score`, `segmentTotals`). The UI, IDE helpers, and export scripts consume these functions as a single source of truth rather than re-implementing their own arithmetic. This ensures that a PHQ-9 total or a segment percentage is identical, whether viewed in the live psychiatry modal, inspected in the Enhanced IDE, or analysed in a research export.
2. **Separation between facts and narratives.** Numeric facts (scale scores, segment durations, lap timestamps) are always produced by the deterministic engine; narratives (clinical notes, formulations, psychoeducation paragraphs) are produced by humans and optionally co-authored with AI. For Persona 1 this means that the AI can draft a note but never "decides" the PHQ-9 total or timer splits. For Persona 2 this means that prompt templates refer to structured fields (e.g. `{{phq9.total}}`) instead of asking the model to infer values from text. For Persona 3 this means that the research dataset is built directly from the structured outputs, regardless of how notes or prompts evolve.
3. **De-identification by design.** Any artefact that can plausibly leave the clinical environment (timer audit HTML, CSV exports, research JSON, AI telemetry) is constructed around pseudo-identifiers and de-identified fields. The identifiers used in `TimerAuditRecord` and in the conceptual `SessionExportRecord` are deliberately decoupled from raw patient identifiers. Persona 1 sees a rich, identifiable clinical surface; Personas 2 and 3, by default, work against safely anonymised abstractions, with re-identification (if ever allowed) handled outside of SynapseCore.
4. **Observability and explainability.** Every non-trivial flow that affects clinical work is observable in two senses: (i) technically, through spans and telemetry emitted by the observability layer; and (ii) clinically, through human-readable summaries such as the timer audit report and MBC overview cards. This allows Persona 2 to confirm that a new feature behaves as intended in production, and allows Persona 3 to reconstruct patterns of care without relying on opaque logs.
5. **Low-friction clinical surface, high-fidelity backend.** For Persona 1, the system presents as a single coherent workspace: patient header, psychiatry modal, MBC cards, AI panel, and timer behave like facets of one tool rather than separate applications. For Personas 2 and 3, the same workspace is revealed as a composition of well-typed modules, explicit interfaces, and exportable records. Internally, the model is closer to a small specialised platform than to a monolithic app: each persona sees a different layer of the same structure.
6. **Backward compatibility for clinical content.** The "legacy" psychiatry modules (e.g. `psychiatry/legacy/mbc.ts` and early AI prompt definitions) remain supported via thin compatibility shims that depend on the modern calculators and export types. This allows existing workflows and teaching materials to coexist with new components built for the Enhanced IDE, without diverging scoring rules or documentation patterns. Persona 1 can thus continue to use familiar cards, while Persona 2 incrementally migrates flows to newer patterns, and Persona 3 can still treat the exports as one continuous

dataset.

Taken together, these invariants ensure that SynapseCore behaves as a single, coherent digital psychiatry workbench across roles. The same timer state, MBC score objects, and AI interactions that support a front-line encounter can be inspected, extended, and analysed without translation errors or hidden side paths. This is essential for a system that aims simultaneously to support day-to-day psychiatry practice, clinical informatics work, and rigorous, reproducible research on session structure and outcomes.

Chapter II: Architecture and Implementation

Sections 3–12

Overview. Part II constitutes the constructive core of the manuscript. It describes the SynapseCore workbench as a layered architecture that realises contributions C_1–C_6 in concrete modules and data structures. Section 3 provides the system-level overview: technology stack, top-level module map, data flow, and integration points. Section 4 introduces the multi-provider AI orchestration engine (C1), including the runtime configuration, model registry, and provider-specific clients expressed through a normalised call interface. Section 5 details the psychiatry knowledge framework (C2), where sections, cards, and content slices form a typed graph that binds structured clinical data to AI prompts and forms. Section 6 formalises the MBC engine (C3) and its autoscore functions, defining scoring schemas, severity bands, and the generation of AI-ready HTML reports and export artefacts. Section 7 presents the flows framework (C4), treating clinical processes as stateful forms with explicit schema and transitions, and describing how AI narrative generation is attached to these states. Section 8 describes the session timer and audit engine (C5), including the timer core, UI, mapping of laps to clinical events, and hooks for session-level machine learning. Section 9 explains the clinical tools and Consulton module, showing how calculators, dashboards, and consultation workspaces are embedded in the Center Panel shell. Section 10 focuses on user interface, layout shells, and the Synapse theme system, covering layout primitives, typography, and theme tokens. Section 11 introduces the Enhanced IDE and developer tooling (C6), including the editor/AI bridges and use cases for local clinical informatics teams. Section 12 describes observability and telemetry, including OpenTelemetry integration, core metrics, AI route instrumentation, and the foundations for future analytics.

Formally, Part II can be viewed as constructing an architecture mapping

$$\mathcal{A} : \mathbf{r} \longmapsto (M_{\mathrm{AI}}, K_{\mathrm{psy}}, E_{\mathrm{MBC}}, F_{\mathrm{flows}}, T_{\mathrm{timer}}, D_{\mathrm{dev}}),$$

where M_{AI} is the orchestration layer, K_{psy} the psychiatry knowledge framework, E_{MBC} the MBC engine, F_{flows} the set of flows, T_{timer} the timer/audit subsystem, and D_{dev} the developer-facing IDE and tooling. The emphasis is on system-level decomposition and the interfaces between these components.

3 System Overview and Architecture

3.1 Technology Stack

SynapseCore is implemented as a browser-based single–page application (SPA) built with React and TypeScript. The design goal is to provide a clinically robust, low-friction workbench for psychiatrists while remaining technically transparent, testable, and easy to extend for new digital psychiatry workflows.

At runtime, the entry point `src/main.tsx` mounts the React tree into a single DOM container, wraps it in a small set of global providers (`ThemeProvider`, routing, and error boundaries), and hands off control to `<AppRoot/>` and the main application shell (`src/App.tsx`, `src/app/AppShell.tsx`). From that point onward, all navigation is client-side: React Router drives URL-based page selection, and feature modules under `src/features/` and `src/centerpanel/` render the different clinical surfaces (home dashboard, psychiatry workbench, timer modal, tools panels, and so on).

React + TypeScript SPA. The entire UI is written in modern React with function components and hooks. TypeScript is used pervasively: component props, store slices, and services all expose explicit types, and most internal helpers are generic over domain entities (for example, timer laps, flows, and case snippets). This gives three concrete advantages for a clinical context:

- **Predictable state evolution.** Components that represent clinically sensitive surfaces (e.g. risk cards, flows, follow-up forms) receive strongly typed props. Invalid combinations of flags and values (such as a "completed" encounter without a recorded risk grade) are prevented at compile time instead of surfacing as runtime bugs.
- **Refactorability for protocol changes.** Changes in clinical protocols (for example, introducing a new severity band or a new flow step) correspond to changes in a small set of TypeScript types. The compiler then points to every location that must be revised to keep the behaviour coherent, which is crucial when adapting the system to different health systems or new guideline revisions.
- **Safe integration of AI helpers.** Provider client code under `src/ai/` uses typed request and response shapes. A prompt that drives a letter generator or psychoeducation helper cannot silently change schema without triggering a type error in the front-end adapter.

Conceptually, the SPA can be seen as a pure function from typed state to a rendered view:

$$\text{RenderedView}_t = f_\theta\Big(S_t^{\text{domain}},\ S_t^{\text{session}},\ S_t^{\text{ui}}\Big), \tag{3.1}$$

where S_t^{domain} encodes the clinical data (current patient, encounter, flows, psychometrics), S_t^{session} encodes transient tools such as the session timer or active AI conversation, and S_t^{ui} encodes pure presentation state (theme, panel layout, open drawers). Keeping these three state families explicit and typed is a central design constraint for the codebase.

Styling and theming with `styled-components`. Layout and visual design are handled with `styled-components`. Global CSS reset and typography rules live in `src/styles/GlobalStyles.ts` and `src/theme/GlobalSynapseStyles.ts`, while concrete theme tokens are defined in `src/theme/` (for example, `synapse.ts` and `np.ts` for the clinical "NP" skin).

The `ThemeContext` in `src/contexts/ThemeContext.tsx` exposes the current theme and a small API to toggle variants (light / dark / clinical high-contrast). Components consume this

via the standard `ThemeProvider`; no component reaches into global CSS directly. This ensures that a psychiatrist can switch the entire workbench from a bright, teaching-lab palette to a calmer, consultation-room palette without breaking layout or contrast guarantees.

Theme tokens such as spacing, radii, and typography weights are further centralised in `src/constants/design.ts`. Clinical components (for example, the risk strip, flows, or timer modal) do not hard-code arbitrary pixel values; they use these tokens instead, which enforces consistent vertical rhythm, hit-area sizes, and font scales across the workbench.

Routing and application shell. Client-side routing is implemented with React Router. `AppRoot` wires `BrowserRouter` and a small route map (see `src/app/AppRoot.tsx` and `src/routes/`): route segments correspond to broad modes of work (home, psychiatry workbench, tools, experimental panels). Under each route, feature modules compose their own internal navigation—for example, the psychiatry workbench uses a split layout with a left rail, center panel, and right panel assembled from files in `src/features/psychiatry/` and `src/centerpanel/CenterPanelShell.tsx`.

The application shell itself is intentionally thin. It is responsible only for top-level layout, theme toggling, and wiring global providers (for example, theme, AI provider configuration, and observability). All domain-specific logic lives in feature modules so that, in principle, a different clinical vertical (e.g. neurology or rehabilitation) could reuse the same shell.

Local stores with Zustand-style patterns. State management uses a light-weight store pattern inspired by Zustand. The state for the application shell and cross-cutting features is implemented via `createSynapseStore` in `src/stores/appStore.ts`, while more specialised store slices live in modules such as `src/centerpanel/state/useTimerEngine.tsx`, `src/centerpanel/Flows/uiStore.ts`, and `src/features/psychiatry/store.ts`.

Each store is defined by:

- a serialisable, typed state object;
- a small set of actions that evolve the state in response to user input or external events (for example, advancing the timer engine, applying a flow template, or pushing an AI suggestion into a draft);
- optional middleware layers (immer for immutable updates, persistence and hydration, and shallow selectors for performance).

The pattern can be written schematically as

$$S_{t+1} = \mathcal{R}(S_t, a_t),$$

where S_t is the store state at time t, a_t is an action (such as "start timer", "mark flow step as done", "append AI suggestion"), and $\mathcal{R}$ is a pure reducer. Because actions are typed and stores are defined in small, feature-scoped files, the resulting code remains easy to audit and to reason about for a clinician–developer.

Several stores, particularly those in `src/stores/` and `src/centerpanel/state/`, use a persistence layer that hydrates from and writes back to `localStorage` via thin wrappers in `src/services/storage.ts`. This allows a psychiatrist to close the browser in the middle of assembling a letter or running a session timer and resume later without losing context, while still keeping all data local to the device.

Observability and telemetry. The stack includes a small observability layer in `src/services/telemetry.ts`. Rather than scattering logging or network calls throughout compo-

nents, clinically relevant events (for example, “session timer started”, “AI draft accepted”, “risk grade changed”) are sent through a single telemetry API. The service can dispatch to the browser console in development or to a configurable endpoint when connected to an observability backend.

This design keeps the clinical UI code free from vendor-specific instrumentation and makes it possible to later attach OpenTelemetry-style tracing without rewriting components. In architectural terms, telemetry is an orthogonal slice of the stack, not a cross-cutting tangle inside the view layer.

Optional TensorFlow.js integration. Machine-learning experiments for timing and workload prediction are isolated in `src/centerpanel/timerHooks/useSessionML.ts`. This hook demonstrates how to attach TensorFlow.js models to the timer engine without contaminating the core clinical logic:

- The core timer state and operations are implemented in `useTimerEngine.tsx`. They are completely independent of any ML library and can be used stand-alone.
- The ML hook subscribes to timer state as an observer: it receives a stream of lap events and aggregate features (for example, elapsed time, number of laps, and segment labels), constructs a feature vector, and passes it to a TensorFlow.js model when one is available.
- All TensorFlow.js imports are localised to this hook. When the hook is not used or when ML is disabled, no TensorFlow.js code is loaded, which keeps the baseline bundle size small and avoids unnecessary complexity for deployments that do not need predictive suggestions.

In functional terms, the ML layer defines an auxiliary mapping

$$\hat{y}_t = g_\phi\big(X_t(S_t^{\text{session}})\big),$$

where X_t extracts simple features from the current session timer state and g_ϕ is a TensorFlow.js model parameterised by ϕ. The prediction $\hat{y}_t$ (for example, a suggested remaining time or a fatigue risk indicator) is treated as a recommendation and surfaced in the UI only as an optional aid; it never replaces explicit clinician judgement.

Layered view of the stack. Putting these elements together, the technology stack can be summarised as the layered structure in Figure 1. Each layer is represented by a small, well-scoped set of directories in the codebase, which keeps the system inspectable for clinicians who also code.

```
Application shell (AppRoot, AppShell)
↓ React Router pages (src/routes/, feature entry points)
↓ Feature modules (src/features/psychiatry, src/centerpanel/)
↓ Stores & services (src/stores/, src/services/)
↓ AI / ML adapters (src/ai/, useSessionML)
↓ Browser platform (React DOM, TypeScript, localStorage, TensorFlow.js)
```

Figure 1: *High-level layering of the SynapseCore front-end technology stack.*

This stack is deliberately conservative: it relies on widely used, well understood web technologies (React, TypeScript, CSS-in-JS, small composable stores) and isolates optional experimentation (AI and TensorFlow.js) so that the clinical workbench remains stable even as individual tools and models evolve.

Root module	Example files	Primary responsibility
AI substrate	`modelRegistry.ts`, `samplerMap.ts`	Declarative model catalogue, provider metadata, and sampling policies.
AI panel and psychiatry AI	`ConsultonPanel.tsx`, `PsychAiBridge.tsx`	Clinical AI panel UI, provider selection, streaming bridge, telemetry.
Psychiatry feature set	`PsychiatryWorkbench.tsx`, `sessionStore.ts`	Domain flows, psychometric calculators, session store, seed content.
Center panel shell	`CenterPanel.tsx`, `SessionTimerModal.tsx`	Workbench layout, timer, tools surface, and clinical shell.
IDE + file system	`IdeShell.tsx`, `FileExplorer.tsx`	Embedded IDE viewport, file explorer, and project tree.
Theme system	`theme.ts`, `GlobalSynapseStyles.ts`	Design tokens, colour system, typography, and global layout rules.
Observability	`otel.ts`, `spans.ts`	OpenTelemetry wiring and internal event/trace definitions.
Application entry	`AppShell.tsx`, `App.tsx`	Root routing, modal orchestration, providers, and bootstrapping.

Table 1: *High-level module map of the SynapseCore codebase.*

3.2 Top-Level Module Map

At the implementation level, SynapseCore is organised as a small number of top-level modules. Each module owns a clearly defined responsibility and communicates with the others through typed props, hooks, and service interfaces rather than ad-hoc imports. Let

$$\mathcal{M} = \{\mathcal{A}_{\text{ai}}, \mathcal{A}_{\text{panel}}, \mathcal{A}_{\text{psy}}, \mathcal{A}_{\text{shell}}, \mathcal{A}_{\text{ide}}, \mathcal{A}_{\text{theme}}, \mathcal{A}_{\text{obs}}, \mathcal{A}_{\text{app}}\}$$

denote the set of these modules. Each $\mathcal{A}_{(\cdot)}$ aggregates a family of React components, hooks, and services under a common root directory, as summarised in Table 1.

AI substrate: `src/ai/`. The AI substrate collects all model- and provider-facing concerns under a single namespace. The model registry expresses each supported model as a typed record (provider id, model id, token limits, and cost estimates), while sampler maps encode consistent decoding behaviour for different clinical tasks (for example, more deterministic sampling for risk summaries, more open-ended sampling for psychoeducation drafts). The rest of the system treats this layer as a pure configuration and capability description, without hard-coding concrete model names inside feature code.

Formally, the AI layer exposes a mapping

$$f_{\text{ai}} : \text{Task} \times \text{Context} \longrightarrow \text{SamplingConfig},$$

which assigns to each clinical task (e.g. “risk summary”, “therapy letter”) and runtime context (provider, maximum cost, session token budget) a concrete sampling configuration that can be passed to a provider client.

AI panel and psychiatry AI: `src/components/ai/`. This module implements the visible AI workbench, bridging the abstract model registry with clinician workflows. The panel components read from the registry, construct provider clients, and orchestrate streaming completion into psychiatry-specific surfaces such as structured notes, risk cards, or psychoeducation sections. Telemetry hooks in this module log clinically meaningful events (for example, “AI draft

accepted" or "grade adjusted after AI suggestion") through the observability layer, rather than emitting ad-hoc console calls.

From the perspective of the rest of the application, this module behaves as a single React component

$$C_{\text{ai}} : (\text{SessionState}, \text{UiSkin}) \longrightarrow \text{JSX},$$

parameterised only by high-level session state and theming information.

Psychiatry feature set: `src/features/psychiatry/`. All domain-specific logic lives under the psychiatry feature module. This includes the high-level workbench component, tab and flow definitions, clinical calculators (for example, MBC metrics, psychometric score aggregation), de-identification helpers, and the dedicated session store. Client code interacts with this module via hooks such as the session store, typed accessors for the current patient/encounter, and pure functions that compute derived metrics from raw questionnaire scores.

Conceptually, this layer realises a mapping

$$f_{\text{psy}} : \text{RawObservations} \longrightarrow \text{ClinicalState},$$

where RawObservations are psychometric items, free-text summaries, and timer events, and ClinicalState is the normalised representation used by both the UI and AI layers.

Center panel shell: `src/centerpanel/`. The center panel module provides the structural shell in which psychiatry flows run. It defines the main three-pane layout, hosts the session timer, and exposes a "tools" rail (Consulton, export, utilities) that can be extended without touching the psychiatry feature code. The shell receives domain state from `src/features/psychiatry/` and delegates AI-related interactions to `src/components/ai/`, acting as an adapter between domain logic and generic workspace chrome.

IDE and file system: `src/components/ide/` and `src/components/file-explorer/`. These modules implement the embedded IDE and file explorer. The IDE shell manages editor tabs, language modes, and basic commands, while the file explorer presents a project tree for workbench-related files (for example, prompt libraries, export templates, and clinical checklists). Both modules are deliberately kept independent from psychiatry domain types so that they can be reused for other clinical verticals or for teaching and research projects.

Theme system: `src/theme/` and `src/ui/theme/`. The theming modules centralise all visual design decisions (colour scales, typography, layout spacing, elevation, and focus treatments). A single theme object is provided at the top of the React tree and consumed by both clinical and IDE components, ensuring that changes in contrast or layout rules propagate consistently across the workbench. Global style files in `src/ui/theme/GlobalSynapseStyles.ts` define base styles for the `body`, scrollbars, and root layout, while `src/theme/theme.ts` holds the tokenised palette and component-level tokens.

Observability: `src/observability/`. The observability module wires the OpenTelemetry configuration and defines the small set of spans and events that are emitted from clinical and AI components. Files such as `src/observability/otel.ts` and `src/observability/spans.ts` provide a thin abstraction over the underlying telemetry client, allowing the rest of the codebase to record events by importing named helpers (for example, `recordAiDraftSpan` or `recordTimerEvent`) instead of manipulating tracer instances directly. This keeps the clinical

UI code free from vendor-specific instrumentation and makes it possible to route telemetry either to the browser console in development or to a remote backend in production.

Application entry: `src/app/` and `src/App.tsx`. Finally, the application entry module composes all the layers above. The root `src/App.tsx` file instantiates the theme provider, telemetry provider, routing, and global modals, then mounts the psychiatry workbench into the router. At a high level, the composition can be written as

$$\mathrm{App} = T_{\mathrm{theme}} \circ T_{\mathrm{telemetry}} \circ R_{\mathrm{router}}\big(C_{\mathrm{workbench}}\big),$$

where T_{theme} and $T_{\mathrm{telemetry}}$ are provider wrappers, R_{router} is the route configuration, and $C_{\mathrm{workbench}}$ is the psychiatry workbench exported from `src/features/psychiatry/`. This composition keeps the clinical modules oblivious to routing and bootstrapping concerns, while still allowing the entry layer to insert global concerns such as error boundaries or future authentication providers.

3.3 High-Level Data Flow

At runtime, most interactions in SynapseCore can be read as instances of a small number of data-flow patterns. Each pattern starts from a user-facing component, passes through a thin hook or service layer, and terminates either in an AI call or in an export artefact. Let UI, State, AI, and Export denote, respectively, the set of visible components, the psychiatry session store, the AI substrate (Section 3.2), and the collection of export recipes. At a high level we can view the system as realising the composition

$$\Phi : \mathrm{UI} \xrightarrow{\phi_{\mathrm{state}}} \mathrm{State} \xrightarrow{\phi_{\mathrm{ai}} \vee \phi_{\mathrm{export}}} \big(\mathrm{AI} \vee \mathrm{Export}\big),$$

where ϕ_{state} records user intent into the psychiatry store, ϕ_{ai} turns selected state into AI prompts and decoding settings, and ϕ_{export} renders state into human-readable clinical documents.

From user action to AI response. The primary AI path begins in the `SynapseCoreAIPanel` component in `src/components/ai/panel/SynapseCoreAIPanel.tsx`. Each user action (typing in the composer, submitting a turn, changing model or temperature, pinning or unpinning items) is first reflected in the local chat state managed by the `useSynapseChat` hook in `src/components/ai/panel/hooks/useSynapseChat.ts`. Conceptually, the hook implements an update function

$$u_{\mathrm{chat}} : (\mathrm{ChatState}, \mathrm{UiEvent}) \longrightarrow \mathrm{ChatState}',$$

where UiEvent ranges over actions such as “append token”, “submit message”, or “toggle system prompt”.

Once a turn is submitted, `useSynapseChat` assembles a concrete AI request (messages, system prompt, context bundle, and decoding settings) and hands it off to the streaming hook `useAiStreaming` in `src/hooks/useAiStreaming.ts`. This hook takes a provider client (constructed from the model registry in `src/ai/modelRegistry.ts`) and exposes a streaming interface

$$f_{\mathrm{stream}} : \mathrm{Request} \longrightarrow \mathrm{TokenStream},$$

where TokenStream is an asynchronous stream of token deltas.

On the rendering side, the `SynapseCoreAIPanel` subscribes to this stream and incrementally merges deltas into the immutable chat transcript rendered by `MessageList` in `src/co`

mponents/ai/panel/MessageList.tsx. In schematic form, the entire AI path can be summarised as

$$\text{UiEvent} \xrightarrow{u_{\text{chat}}} \text{Request} \xrightarrow{f_{\text{ai}}} \text{SamplingConfig} \xrightarrow{f_{\text{stream}}} \text{TokenStream} \xrightarrow{r_{\text{render}}} \text{ChatTranscript},$$

where f_{ai} is the task–context mapping introduced in Section 3.2 and r_{render} denotes the pure reducer that the panel uses to fold streaming updates into a list of messages.

From psychiatry content selection to AI context. The AI panel itself is intentionally agnostic about the details of the psychiatry workbench. Clinical semantics enter the pipeline through selection state managed by the psychiatry feature module in `src/features/psychiatry/`. Components such as `PsychiatryModal` (`src/features/psychiatry/PsychiatryModal.tsx`) allow the clinician to select flows, cards, or specific sections (for example, "risk factors", "MSE – mood", or a particular psychometric scale). These selections are written into the central psychiatry store via the `usePsychStore` hook in `src/features/psychiatry/store.ts`.

Downstream, the AI context builder in `src/lib/ai/context.ts` materialises a *context bundle*, i.e. a normalised snapshot of the selected clinical state plus relevant metadata. The core function `buildContextBundle` can be viewed as

$$f_{\text{ctx}} : \text{ClinicalState} \times \text{SelectionSpec} \longrightarrow \text{ContextBundle},$$

where ClinicalState is the structured session representation derived from questionnaires, notes, and flows, and SelectionSpec describes which parts of this state are to be exposed to the AI layer. The resulting ContextBundle (containing items such as current diagnoses, MBC metrics, and selected narrative sections) is injected into the AI request that `useSynapseChat` hands off to `useAiStreaming`. In this way, the psychiatry module controls *what* the AI sees, while the AI substrate controls *how* the model is called.

From flows and timer events to exports. A second major data path flows from the same psychiatry and timer state into export recipes. Flows under `src/features/psychiatry/Flows/` (assessment checklists, intervention forms, psychoeducation templates) and the session metadata (segments and laps recorded by the timer modal in `src/centerpanel/components/TimerModal.tsx`) are all persisted in the psychiatry store. The tools rail in the centerpanel shell exposes this state to export components such as `ConsultonPanel` and `ExportBar` in `src/centerpanel/Tools/`.

Export logic is factored into small, composable recipes located in `src/centerpanel/Tools/exportRecipes/`. For example, the `consult` recipe in `src/centerpanel/Tools/exportRecipes/consult.ts` assembles a referral-style consult letter by pulling from MBC metrics, risk fields, and selected narrative blocks. The shared assembly helper `assembleForPreview` in `src/centerpanel/Tools/lib/assemble.ts` realises a mapping

$$f_{\text{export}} : (\text{ClinicalState}, \text{TimerTrace}) \times \text{ExportRecipe} \longrightarrow \text{ExportDoc},$$

where TimerTrace encodes segment timing and laps, and ExportDoc is an opaque representation of the export body (currently rendered as rich text in the tools panel, with options to copy to the clipboard or save to a file).

From the perspective of a clinician, this entire path collapses to a single interaction: complete flows and let the timer run during the session; open the tools rail; select an export recipe; and review or edit the generated draft. Internally, however, the same normalised ClinicalState drives both the AI panel and the export system, ensuring that clinical facts are recorded once and then reused coherently across AI-assisted drafting, progress notes, and external letters.

3.4 External Dependencies and Integration Points

SynapseCore treats every external dependency as a *bounded integration surface* rather than as an opaque black box (Vrdoljak et al., 2024). In digital psychiatry this is critical for at least three reasons: (i) large language models (LLMs) and other AI services must be callable in a predictable and auditable way (Gaber et al., 2025; Maity & Saikia, 2025b; Nazi & Peng, 2024), (ii) on-device learning components must not leak clinical context or identifiers (Connolly et al., 2021; Philippe et al., 2022; Ridout et al., 2024; TensorFlow.js Team, 2019), and (iii) observability must be strong enough to surface performance, reliability, and safety issues early (OpenTelemetry Project, 2025; OpenTracing Project, 2023).

Formally, we view the running system at any time as a tuple

$$\mathcal{S}(t) = \big(\mathcal{U}(t), \mathcal{C}(t), \mathcal{R}(t), \mathcal{D}(t)\big),$$

where $\mathcal{U}(t)$ encodes user state and actions (clicks, form entries, timer laps), $\mathcal{C}(t)$ the internal clinical context (scales, flows, content selection), $\mathcal{R}(t)$ the active AI route (provider, model, sampling configuration), and $\mathcal{D}(t)$ the set of active external dependencies and their health status (e.g., AI provider reachability, telemetry exporter availability). The goal of the integration layer is to guarantee that all calls from $\mathcal{C}$ to $\mathcal{R}$ and $\mathcal{D}$ are mediated by typed, testable adapters with explicit failure-handling and observability hooks.

We distinguish three families of external dependencies:

1. AI provider APIs (OpenAI, Anthropic, Gemini, Ollama, and institution-specific/custom endpoints).
2. On-device learning via TensorFlow.js for session pattern learning and local experimentation.
3. Observability and tracing via OpenTelemetry-compatible exporters.

The subsections below describe these dependencies, their associated models and tasks, and the way they are exposed to the rest of the system.

3.4.1 AI provider integrations and model registry

The AI orchestration engine (Section 4) exposes providers and models through a unified *route* abstraction. Let $\mathcal{P}$ denote the set of providers and $\mathcal{M}_p$ the set of models available for provider $p \in \mathcal{P}$. A route is defined as

$$r = (p, m, s) \in \mathcal{R} = \bigcup_{p \in \mathcal{P}} \big(\{p\} \times \mathcal{M}_p \times \mathcal{S}\big),$$

where $s \in \mathcal{S}$ is a sampling configuration (temperature, top-p, maximum tokens, JSON mode flags, etc.).

At runtime, the application presents clinically meaningful labels for routes (e.g., *"OpenAI GPT-4.1 – structured"*, *"Anthropic Claude 3.x – long-form"*) rather than raw model IDs, but the underlying configuration remains fully explicit and serializable. The model registry defines the static subset of supported models per provider and can be extended with institution-specific or research-only models. Table 2 summarizes the current classes of models as they are exposed in the user interface.

Table 2: *Scientific comparison of AI providers and model families integrated into SynapseCore.*

Provider	Model family / examples	Modalities & context	Governance & safety	Primary clinical use	Integration notes
OpenAI	GPT-4.1, GPT-4o, GPT-4o-mini	Text, code, limited vision; 128k–200k ctx	High-governance; strong filters; reliable JSON mode	MSE narratives, DX scaffolds, structured notes	Stable latency; robust function-calling; precision text
Anthropic	Claude 3.x family	Text; ultra-long context (200k–1M)	Highest caution tier; guideline-aware	Long-form summaries, psychotherapy letters	Excellent coherence; ideal for sensitive flows
Google (Gemini)	Gemini 1.5 / 2.x	Text+vision multimodal; 128k ctx	Research-tier; moderate governance	Multimodal timelines, labs+images+text synthesis	Best for prototypes; not for high-risk content
Ollama / Local	LLaMA, Mistral, tuned local LLMs	Text, code; context varies	Local-governed; data stays on device	Offline drafting, protocol tuning, style transfer	No external egress; version-dependent reproducibility
Custom endpoints	Institutional fine-tuned LLMs	Text, embeddings; 8k–64k ctx	Institutional governance; IRB/privacy constraints	EMR-specific summarization, risk scoring	Access-controlled; isolated telemetry; monitoring required

From a mathematical perspective, each provider–model pair is treated as a probabilistic transducer

$$\mathcal{M}_{p,m} : (\Sigma^*, C) \to \mathcal{D}(\Sigma^*),$$

mapping an input token sequence and a context bundle C (e.g., scales, flow state, section snippets) to a distribution over output token sequences. The orchestration layer normalizes prompts into

$$x = \Phi(u, C, r),$$

where u is raw user input, C is typed contextual data, and r is the active route. This separation allows us to adjust provider-specific parameters (such as safety settings and JSON mode) without rewriting clinical prompts.

From a clinical governance perspective, different provider classes are used for different tasks. For example, models with stronger retrieval-augmented reasoning are reserved for complex formulation tasks and treatment planning, whereas lighter, faster models are used for quick phrasing refinements or psychoeducation drafts where the clinician is clearly in the loop. This follows emerging recommendations on tiered use of LLMs in healthcare, where high-risk tasks either remain unsupported or require stricter oversight, while lower-risk documentation support is more permissive (Gaber et al., 2025; Li et al., 2025; Maity & Saikia, 2025b; Nazi & Peng, 2024).

3.4.2 On-device session pattern learning (TensorFlow.js)

In addition to server-side AI models, SynapseCore includes an optional on-device learning component implemented in TensorFlow.js. Its scope is intentionally narrow: local models are used for *session pattern learning* and user-experience optimisation rather than for direct diagnostic or treatment recommendations.

Let $\mathcal{S}_{\text{sess}}$ denote the space of session summaries. Each session k is represented as a feature vector

$$\phi(s_k) = \left[t_k^{\text{start}},\, \text{DoW}_k,\, T_k^{\text{total}},\, n_k^{\text{laps}},\, \vec{d}_k,\, \vec{p}_k\right] \in \mathbb{R}^d,$$

where t_k^{start} is the (normalized) start time, DoW_k the day of week, T_k^{total} the total duration, n_k^{laps} the number of laps, $\vec{d}_k$ the vector of durations per segment (e.g., check-in, assessment,

intervention, documentation), and $\vec{p}_k$ aggregates pauses and overruns. Local models f_θ are trained purely on these non-identifiable features:

$$\hat{y}_k = f_\theta\big(\phi(s_k)\big),$$

where $\hat{y}_k$ might encode, for example, the predicted probability that a planned 50-minute session will overrun, or the recommended choice of preset for the next consultation.

Table 3 lists the main tasks currently envisioned for TensorFlow.js within the system.

Table 3: *Representative TensorFlow.js tasks in SynapseCore. All features are derived from timer events and high-level session metadata; no raw text, scales, or identifiers are used.*

Task	Input feature examples	Output / prediction	Intended effect
Session duration prediction	Start time, weekday, recent session lengths, segment mix, previous overruns	Expected total duration; overrun probability; risk of late-session compression	Suggest realistic countdown presets; improve pacing; reduce late-session rush
Segment pacing support	Segment durations, order across previous sessions, average documentation lag	Probability that documentation will be postponed or squeezed into final minutes	Encourage earlier documentation; promote mid-session notes; reduce post-session backlog
Fatigue / overload proxy	Sessions per day, time gaps, late-evening load, cumulative weekly duration patterns	Overload tier classification (low/medium/high)	Nudge clinician toward breaks; warn about clustering; support sustainable scheduling

Because all training happens in-browser and the data remain on the clinician's machine, this design aligns with privacy-preserving principles in digital mental health (Philippe et al., 2022; Putica et al., 2025; Ridout et al., 2024). The feature map ϕ is intentionally limited in scope, and the persistence layer can be disabled entirely if the deployment environment wishes to avoid any local learning.

3.4.3 Observability and OpenTelemetry-compatible tracing

Modern digital health tools require not only functional correctness but also *observability*: a structured view of what the system is doing and how it behaves under load (OpenTelemetry Project, 2025; OpenTracing Project, 2023; Sandberg, 2024). SynapseCore adopts the OpenTelemetry data model for traces, metrics, and logs, and maps its own internal events onto three main span families:

- **AI spans** for provider calls (including route ID, token usage estimates, latency, and error flags).
- **Interaction spans** for high-level user actions (opening psychiatry modal, starting a flow, starting/pausing the timer).
- **Export spans** for Consulton and tools (generating notes, letters, CSV/Markdown exports).

If we denote by $\mathcal{E}$ the event stream generated by the application and by Ψ the mapping to OpenTelemetry spans, we can write

$$\Psi : \mathcal{E} \to \mathcal{T},$$

where $\mathcal{T}$ is the set of telemetry records (spans and metrics) shipped to the configured backend (or kept in-memory during development). For an AI call event e^{AI}, the mapping annotates the span with:

$$\Psi(e^{\mathrm{AI}}) = \big(\text{route\_id} = r, \quad \text{tokens\_input} = n_{\text{in}}, \quad \text{tokens\_output} = n_{\text{out}}, \quad \text{latency} = \Delta t, \quad \text{status} \in \{\text{OK, ERROR, ABORTED}\}\big). \tag{3.2}$$

This representation supports both engineering-oriented questions (e.g., *"Which routes have the highest tail latency?"*) and informatics questions (e.g., *"How often is the risk narrative generator aborted compared to other prompts?"*). Table 4 summarizes the principal observability signals.

Table 4: *Key OpenTelemetry-style signals emitted by SynapseCore for each external dependency family.*

Dependency family	Span attributes	Derived metrics	Clinical relevance
AI providers	route ID, provider name, model ID, sampling preset, latency, token counts, error code	Per-route latency distribution, error rate, approximate cost or usage proxies	Detect degraded providers; surface instability in AI-assisted risk summaries or formulation tasks
TensorFlow.js session ML	task type, feature dimension, training time, model version, convergence tokens	Training frequency, convergence time, model size, update cadence	Ensure local learning remains lightweight; detect stalls requiring model reset; maintain responsiveness
Exports / Consulton	export type, template ID, output size, rendering time, generation backend	Export throughput, failure count, template usage distribution	Identify widely used templates; monitor pipeline stability; detect regressions in export performance

Importantly, telemetry is configured to avoid recording any raw clinical text or patient identifiers; only aggregate indicators (token counts, timing, route IDs) and structural metadata are exported. This is consistent with recommendations that observability tooling for health applications focus on meta-level behaviour rather than on content-level data wherever possible (OpenTelemetry Project, 2025; OpenTracing Project, 2023).

3.4.4 Failure modes, graceful degradation, and fallbacks

External dependencies in healthcare workflows will inevitably fail: network outages, rate limits, model deprecations, or telemetry backend downtime. SynapseCore therefore treats the external dependency set $\mathcal{D}(t)$ as a first-class object with an associated health function

$$h : \mathcal{D}(t) \to \{\text{HEALTHY}, \text{DEGRADED}, \text{UNAVAILABLE}\}.$$

For AI providers, health is based on recent spans and error codes. When $h(p) =$ DEGRADED or UNAVAILABLE, the system can:

- Suggest alternative routes (e.g., switching from a heavy model to a smaller but more available one).
- Fall back to purely structured, non-AI workflows (flows, scales, manual templates) without blocking data entry.
- Highlight in the interface that suggestions are currently unavailable or limited.

For TensorFlow.js components, failure or unavailability simply disables local learning features; the core timer still functions as a deterministic state machine. For OpenTelemetry exporters, if the backend is unavailable, spans may be kept in memory for a short period or dropped; in all cases, clinical workflows continue to work offline. This design ensures that the clinician is never prevented from documenting or viewing critical information due to non-essential external dependencies, an important principle in safety-critical systems (Philippe et al., 2022; Ridout et al., 2024).

3.4.5 Integration schema

Figure 2 summarizes the role of external dependencies around the core SynapseCore runtime. The inner rectangle represents the clinician-facing shell (psychiatry content, flows, timer, tools), while the outer layer contains AI providers, local TensorFlow.js models, and observability exporters.

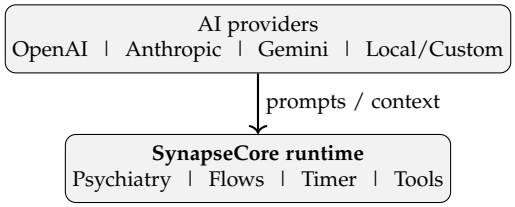


Figure 2: *Schema A: Upstream view showing AI providers feeding prompts and structured model outputs into the SynapseCore runtime.*

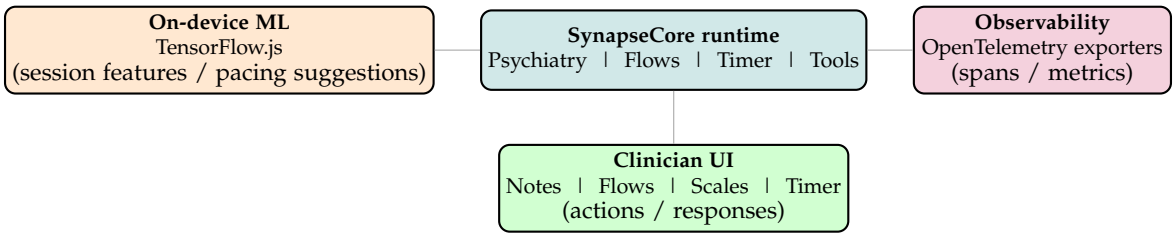


Figure 3: *Compact horizontal integration schema for SynapseCore, with tightly spaced components, no arrows, and color-coded modules.*

4 AI Orchestration Engine

4.1 Runtime Configuration and Model Registry

The AI orchestration engine is parameterised by a normalised runtime configuration object and a model registry, as sketched in Section 4. Concretely, this layer is implemented via (i) a persisted Zustand store `useAiConfigStore` in `src/stores/useAiConfigStore.ts` (typed by `src/stores/useAiConfigStore.types.ts`), and (ii) the model registry utilities in `src/ai/modelRegistry.ts` and `src/ai/modelMeta.ts`. Together they provide a single source of truth for the active provider, model, sampling hyperparameters, API credentials, and coarse provider capabilities, and they expose a minimal `RuntimeConfig` object that downstream components can consume without depending on UI-level state.

Formally, let $\mathcal{P}$ denote the finite set of supported providers,

$$\mathcal{P} = \{\text{openai}, \text{anthropic}, \text{gemini}, \text{ollama}, \text{custom}\},$$

and for each $p \in \mathcal{P}$ let $\mathcal{M}_p$ denote the set of models that can be addressed under that provider. As introduced in Section 4, the model registry realises a mapping

$$\mathcal{R} : \mathcal{P} \times \mathcal{M} \to \mathcal{C},$$

where $\mathcal{M} = \bigcup_{p \in \mathcal{P}} \mathcal{M}_p$ and $\mathcal{C}$ is the space of normalised configuration objects. Each configuration $c \in \mathcal{C}$ is a tuple

$$c = (m, \theta, \sigma, \kappa, \mathbf{s}),$$

where $m \in \mathcal{M}$ is a model identifier, θ encodes sampling hyperparameters, $\sigma \in \{0, 1\}$ indicates whether streaming is enabled, κ is a credential bundle (API key and optional base URL), and $\mathbf{s}$ collects provider-specific metadata mapped into a shared schema (capabilities and HTTP headers).

4.1.1 Configuration state in `useAiConfigStore`

At implementation level, the configuration state is governed by the `AiConfigState` interface and a small set of internal controls in `useAiConfigStore`. The core types are:

- `ProviderId`: a tagged union `'openai' | 'anthropic' | 'gemini' | 'ollama' | 'custom'`.
- `ModelId`: a normalised model identifier (`string`), e.g. `"gpt-4o-mini"` or `"gemini-1.5-pro"`.
- `Sampling`: a structured decoding configuration with fields `temperature`, optional `top_p`, `max_tokens`, a boolean `json_mode`, and an optional `system_prompt`.
- `ProviderKey`: per-provider credentials with optional `apiKey` and `baseUrl`.
- `ProviderCaps`: a coarse capability descriptor with booleans for `streaming`, `jsonMode`, `supportsTopP`, and an optional `tokenLimit`.

The persisted store holds both user-facing choices and derived auxiliary state. At time t we can summarise the AI configuration state as

$$S_{\mathrm{AI}}(t) = \big(p_t, m_t, \theta_t, K_t, L_t, F_t, C_t, Q_t, v_t, r_t\big),$$

where

- $p_t \in \mathcal{P}$ is the active provider (`provider`);
- $m_t \in \mathcal{M}_{p_t} \cup \{\bot\}$ is the selected model (`model`, with $\bot$ for "no model");
- θ_t is the current sampling configuration (`sampling`);
- K_t is the map from providers to `ProviderKey` instances (`keys`);
- L_t is the per-provider model list (`modelList: Record<ProviderId, ModelId[]>`);
- F_t is the per-provider favourite list (`favorites`);
- C_t is the per-provider capability map (`caps: Record<ProviderId, ProviderCaps>`);
- Q_t is the per-provider key verification status (`keyStatus`);
- $v_t \in \mathbb{N}$ is a monotonically increasing configuration version (`configVersion`);
- $r_t \in \mathbb{N}$ stores the model registry version last seen by the store (`registryVersion`).

The store is persisted under the key `synapse.aiConfig.v3` and uses a versioned migration function so that older layouts (for example when the provider previously called `'google'` was renamed to `'gemini'`) are upgraded in-place. On rehydration, if `registryVersion` does not match the current registry constant `__MODEL_REGISTRY_VERSION`, the store walks through all providers, unions their persisted model lists with the current static seeds from the registry, and updates `modelList` and `registryVersion`. This ensures that users automatically see new models added to `STATIC_MODELS` without losing custom entries.

Table 5: *Core runtime configuration schema in SynapseCore.*

Symbol	TypeScript field(s)	Interpretation
p	`provider: ProviderId`	Active AI provider.
m	`model: ModelId | null`	Selected model identifier (or none).
θ	`sampling: Sampling`	Decoding hyperparameters (`temperature`, `top_p`, `max_tokens`, `json_mode`, `system_prompt`).
κ	`keys[provider]: ProviderKey`	Credentials: API key and optional base URL.
σ	`streaming: boolean`	Streaming on/off flag in `RuntimeConfig`.
$\mathbf{s}$	`caps[provider], headers`	Provider capabilities and any extra headers.

Configuration changes are driven by a small set of pure update operations (`setProvider`, `setModel`, `setSampling`, `setKey`, `refreshModels`, `refreshKeyStatus`). The invariants enforced by these operations can be summarised as follows:

$$m_t \in \mathcal{M}_{p_t} \cup \{\bot\}, \tag{4.1}$$

$$L_t(p) \supseteq \mathcal{M}_p^{\text{static}} \quad \text{for all } p \in \mathcal{P}, \tag{4.2}$$

$$v_{t+1} = \begin{cases} v_t + 1, & \text{if the effective configuration tuple } c_{t+1} \neq c_t, \\ v_t, & \text{otherwise,} \end{cases} \tag{4.3}$$

where c_t denotes the induced runtime configuration (defined below). If `setModel` is called with a model that does not belong to the active provider's list $L_t(p_t)$, the call is rejected, a diagnostic event is emitted, and only the version counter may be incremented; the store never admits a cross-provider model mismatch.

Table 5 links the mathematical notation from the previous section to the concrete TypeScript fields that make up the `RuntimeConfig` object.

Clinically, this design keeps the AI configuration surface predictable: only a small and well-typed set of controls can influence generation behaviour, and these controls are shared across all AI-backed tools in the system (chat assistant, note generator, export studio).

4.1.2 Static registry, dynamic discovery, and capabilities

The model registry in `src/ai/modelRegistry.ts` provides both a static seed list and a dynamic discovery mechanism. The constant `STATIC_MODELS` is a map

$$\texttt{STATIC\_MODELS} : \mathcal{P} \to \mathcal{P}(\mathcal{M}),$$

which lists a curated set of models per provider. For example, the OpenAI bucket includes general-purpose and reasoning models such as `gpt-5`, `gpt-5-mini`, `gpt-4.1`, `gpt-4.1-mini`, `gpt-4o`, `o3-mini`, `o1-mini`, and compatible legacy entries. These static seeds ensure that a useful configuration is always available even when dynamic listing fails (e.g. offline operation or temporary provider outage).

Dynamic discovery is handled by the asynchronous `listModelsDynamic(p, keys)` function. For each provider $p \in \mathcal{P}$, a provider-specific client (`src/ai/providerClients/*.ts`) is used to call the relevant model listing endpoint:

- OpenAI: `GET /v1/models` with bearer token;
- Anthropic: `GET /v1/models` with `x-api-key` header;
- Gemini: `GET /v1beta/models` with API key in the query string;
- Ollama: `GET /api/tags` on the local Ollama host.

The returned IDs are normalised via a small utility so that, for example, Gemini identifiers of the form `models/gemini-1.5-pro` are turned into bare `gemini-1.5-pro`. The function

Table 6: *Provider capability matrix (excerpt from* `CAPS`*).*

Provider	Streaming	JSON mode	Top-p	Token limit (approx.)
OpenAI	yes	yes	yes	128 000
Anthropic	yes	no	yes	200 000
Gemini	yes	no	yes	1 000 000
Ollama	yes	no	yes	8 192
Custom	yes	no	yes	8 192

maintains a global cache with key

$$\text{cacheKey} = \text{provider} \parallel \text{baseUrl} \parallel \mathbf{1}_{\{\text{apiKey present}\}},$$

and a time-to-live of 60 s; repeated calls within the TTL re-use the cached list. In symbols, if $D_p(t)$ denotes the dynamic model list for provider p at time t, we have

$$D_p(t + \Delta t) = D_p(t) \quad \text{whenever } \Delta t < 60\,\text{s},$$

subject to cache eviction; beyond the TTL a fresh call is issued.

Within the store, the method `refreshModels` merges static and dynamic models:

$$L_t(p) = \mathcal{M}_p^{\text{static}} \cup D_p(t),$$

and then re-resolves the active model via a helper `resolveModel`. This helper preserves the current model if still present, otherwise falls back to the first available model in $L_t(p)$, and if that is empty, to the registry's notion of a default model for p.

The capability matrix

$$\text{CAPS} : \mathcal{P} \to \texttt{ProviderCaps}$$

encodes coarse provider-level constraints: whether streaming is supported, whether JSON-mode style outputs are available, whether top-p sampling is meaningful, and an approximate context-window limit. The current values are:

These caps are not hard clinical constraints, but they inform the sampling mapper (Section 4.3) when deciding how aggressively to truncate prompts or whether it is safe to request JSON-native responses from a given provider.

4.1.3 Runtime projection and memoisation

Downstream components never read the full `AiConfigState`. Instead they consume a narrowly scoped `RuntimeConfig` projection, defined in `src/stores/useAiConfigStore.types.`

ts as

```
RuntimeConfig = { provider: ProviderId,
                  model: ModelId | null,
                  apiKey?: string,
                  baseUrl?: string,
                  headers?: Record<string,string>,
                  temperature?: number,
                  topP?: number,
                  maxTokens?: number,
                  jsonMode?: boolean,
                  streaming: boolean }.
```

The selector `AiSelectors.runtime` implements a pure projection

$$g_{\mathrm{rt}} : S_{\mathrm{AI}} \longrightarrow \mathcal{C},$$

which computes the minimal configuration needed to build HTTP requests. To avoid unnecessary React re-renders, it maintains an internal signature function

$$\mathrm{sig}(S_{\mathrm{AI}}) = \big(p, m, \mathbf{1}_{\{\mathrm{apiKey}\}}, \mathrm{baseUrl}, \theta\big),$$

serialised into a string. If sig does not change between calls, the previous `RuntimeConfig` object is returned by reference; otherwise a new object is constructed and cached. This memoisation guarantees that downstream effects trigger only when there is a semantics-relevant configuration change.

From a clinical-systems perspective, this separation is crucial: the full configuration store can evolve (for example, adding new favourite models or updating key verification state) without injecting noise into the AI pipeline. Only changes to the effective tuple $(p, m, \theta, \kappa, \sigma, \mathbf{s})$ propagate into the orchestration layer, which keeps AI behaviour stable and predictable across the psychiatry tools built on top of SynapseCore.

4.2 Provider Clients and Normalized Calls

While the runtime store (Section 4.1) determines *which* provider–model pair should be used, the actual HTTP requests are constructed by a thin normalization layer. This layer has two main responsibilities:

1. expose a single, provider-agnostic interface for building both streaming and non-streaming requests; and
2. map the generic sampling configuration θ to the concrete fields required by each provider API, including JSON-mode and other structured-output options.

Implementation-wise, this layer consists of:

- the `BuiltProviderRequest`, `ProviderRequestBuilder`, and `SanitizedRequestSnapshot` types in `src/ai/types.ts`;
- the provider-specific request builders in `src/ai/samplingMapper.ts`; and
- the model-listing clients in `src/ai/providerClients/openai.ts`, `src/ai/providerClients/anthropic.ts`, `src/ai/providerClients/google.ts` (Gemini), and `src/ai/providerClients/ollama.ts`, which are also reused by the model registry.

4.2.1 Normalized request representation

Let

$$(p, m, \theta, x, \kappa, b, s)$$

denote a concrete call configuration, where p is the provider (`ProviderId`), m the model identifier, θ the generic sampling configuration from Section 4.1, x the final prompt text, κ the resolved credentials (API key), b an optional base URL override, and $s \in \{0, 1\}$ the desired streaming flag. The normalization layer realises a mapping

$$B : (p, m, \theta, x, \kappa, b, s) \longrightarrow r,$$

where r is a `BuiltProviderRequest` with the following shape:

```
r = { provider: ProviderId,
      model: string,
      request: { url: string, method: 'POST',
          headers: Record<string,string>, body: any },
      meta: { jsonModeApplied: boolean,
          topPSupported: boolean,
          usedSystemPrompt: boolean,
          estimatedInputChars: number } }.
```

The core builder function `buildProviderRequest(params, opts)` takes a `BuildParams` record—which collects `provider`, `model`, `sampling`, `prompt`, `apiKey?`, and `baseUrl?`—together with an optional `BuildOptions` flag `{ stream?: boolean }`. It then dispatches to a provider-specific `ProviderRequestBuilder`:

$$B(p, \cdot) = \begin{cases} B_{\text{openai}} & \text{if } p = \text{openai}, \\ B_{\text{anthropic}} & \text{if } p = \text{anthropic}, \\ B_{\text{gemini}} & \text{if } p = \text{gemini}, \\ B_{\text{ollama}} & \text{if } p = \text{ollama}, \\ B_{\text{custom}} & \text{if } p = \text{custom}. \end{cases} \tag{4.4}$$

The streaming flag is handled uniformly: each builder receives `opts?.stream` and defaults to streaming when it is omitted,

$$s = \begin{cases} 1, & \text{if } \texttt{opts.stream} \text{ is undefined}, \\ \texttt{opts.stream}, & \text{otherwise}. \end{cases}$$

This flag is then forwarded into the provider-specific request body (e.g. `stream: true` in OpenAI, Anthropic, Gemini, and Ollama payloads, and in the generic `custom` payload). The downstream streaming pipeline (Section 4.4) therefore only needs to inspect `opts.stream` and the provider caps matrix, not raw HTTP parameters.

For debugging and observability, each built request is convertible to a `SanitizedRequestSnapshot` via `sanitizeBuiltRequest`. This helper replaces any `Authorization` and `xapikey` headers with placeholders before the snapshot is written to logs or passed to the UI. The function can be viewed as a projection

$$S : \texttt{BuiltProviderRequest} \longrightarrow \texttt{SanitizedRequestSnapshot},$$

which preserves URL, body, and high-level meta-data while redacting sensitive fields.

Table 7 summarises the key components of the normalized request representation.

Table 7: *Normalized provider request structure in SynapseCore.*

Component	TypeScript field(s)	Role in the system
Provider + model	`provider: ProviderId; model: string`	Identifies the concrete AI route (for example OpenAI GPT-5, Anthropic Claude 3.5, Gemini 1.5 Pro, or a local Ollama model).
HTTP request envelope	`request: { url, method, headers, body }`	Fully specified HTTP call that can be passed to the streaming engine without further provider-specific logic.
Sampling meta-data	`meta.jsonModeApplied, meta.topPSupported, meta.usedSystemPrompt`	Records how the generic sampling configuration was interpreted (for example whether JSON mode was activated and whether a system prompt was injected).
Size estimate	`meta.estimatedInputChars`	Rough character-level estimate used by telemetry and future budgeting logic for prompt length.
Sanitized snapshot	`SanitizedRequestSnapshot`	Redacted view used in logs, teaching demonstrations, and troubleshooting, without exposing API keys.

4.2.2 Per-provider mapping of sampling parameters

The core function of each `ProviderRequestBuilder` B_p is to translate the generic sampling configuration $\theta = (\tau, p_{\text{top}}, n_{\max}, j)$—temperature, top-$p$, maximum output tokens, and JSON-mode flag—into concrete API fields.

For OpenAI, the builder `buildOpenAI` constructs a `/chat/completions` request with the following mapping:

- the system prompt (if present) becomes a `role: 'system'` message, followed by a single `role: 'user'` message containing x;
- τ maps to `temperature`;
- p_{top} maps to `top_p` when specified;
- $n_{\max}$ maps to `max_tokens`;
- $j = 1$ activates `response_format = { type: 'json_object' }`.

For Anthropic, the builder `buildAnthropic` targets the `/messages` endpoint and uses a single user message whose content concatenates the system prompt and user text. The mapping is:

- $\tau \mapsto$ `temperature`, $p_{\text{top}} \mapsto$ `top_p`, $n_{\max} \mapsto$ `max_tokens`;
- if $j = 1$, a `metadata.json_mode = true` flag is attached to the request, so that downstream logic can interpret the output as structured JSON if the model cooperates.

For Google Gemini, the builder `buildGemini` calls `models/{model}:generateContent` on the `v1beta` API:

- the system prompt and user text are merged into a single `contents[0].parts[0].text` field;
- τ and $n_{\max}$ are mapped to `generationConfig.temperature` and `generationConfig.maxOutputTokens`, respectively;
- p_{top} is mapped to `generationConfig.topP` when present;
- $j = 1$ triggers `generationConfig.responseMimeType ='application/json'`.

For local models served via Ollama, `buildOllama` issues a `/api/chat` request:

- the system prompt (if any) is mapped to a `role: 'system'` message in `messages`;
- x becomes the final `role: 'user'` message;
- τ and $n_{\max}$ are mapped to `options.temperature` and `options.num_predict`;
- p_{top} becomes `options.top_p`;
- $j = 1$ results in `format: 'json'` in the request body.

Table 8: *Per-provider mapping of generic sampling parameters to HTTP request fields.*

Provider	Endpoint (base URL omitted)	Temperature / top-p	Max tokens / length	JSON-mode and streaming
OpenAI	`/v1/chat/completions`	`temperature` and optional `top_p` fields on the root request.	`max_tokens` on the root request.	`response_format = {type: 'json_object'}` when `sampling.json_mode` is true; `stream` boolean on the root request.
Anthropic	`/v1/messages`	`temperature` and optional `top_p` on the root request.	`max_tokens` on the root request.	`metadata.json_mode = true` when JSON mode is requested; `stream` boolean on the root request.
Gemini (Google)	`/v1beta/models/{...}:generateContent`	`generationConfig.temperature` and, when present, `generationConfig.topP`.	`generationConfig.maxOutputTokens`.	`generationConfig.responseMimeType = 'application/json'` for JSON mode; `stream` flag field in the payload.
Ollama	`/api/chat` on the local host	`options.temperature` and optional `options.top_p`.	`options.num_predict`.	`format = 'json'` when JSON mode is requested; `stream` boolean on the root request.
Custom	Integrator-defined endpoint URL	Passed transparently inside `sampling`.	Passed transparently inside `sampling`.	JSON mode expressed through the same `sampling` structure; `stream` flag passed as-is.

Finally, for the generic `custom` provider, `buildCustom` constructs a minimal payload:

```
{ model, prompt, sampling, stream },
```

leaving the exact interpretation of θ to the integrator's own endpoint. This design allows clinical deployments to plug in in-house models or third-party gateways while still benefiting from the same UI and runtime configuration machinery.

Table 8 summarises these mappings.

4.2.3 Structured outputs and tool-like responses

Although none of these providers expose a strict "tool calling" API in the normalization layer itself, the JSON-mode mappings above are chosen to encourage *tool-like* responses. When `sampling.json_mode` is enabled, the psychiatry tools (Section 9) request models to emit structured JSON objects instead of free text. In practice this is used for:

- structured risk summaries (e.g. a list of suicide risk factors with confidence scores);
- templated clinical note fragments for insertion into the editor;
- checklists and banded scores aligned with the measurement-based care engine (Section 6);
- export-ready artefacts rendered into FHIR-like or CSV structures.

Because JSON-mode is recorded explicitly in the `meta` field of `BuiltProviderRequest`, downstream components can branch their parsing logic based on whether a response is expected to be plain text or structured data. This combination of a normalized call interface, per-provider parameter mapping, and explicit JSON-mode semantics allows SynapseCore to support a heterogeneous mix of AI providers while presenting a consistent and clinically safe interface to the psychiatrist.

Step 1 — RuntimeConfig & Sampling
`provider`, `model`, `temperature`, top-p, `maxTokens`, `jsonMode`
Clinician sets provider/model and generic decoding options in the shared runtime configuration store.

Step 2 — Normalized builder
`buildProviderRequest( BuildParams, BuildOptions )`
`BuildParams` collects the runtime fields (provider, model, sampling, prompt, API key, base URL).
`BuildOptions` carries call-level flags (for example `stream: boolean`).
The builder produces a single `BuiltProviderRequest` with a fully specified HTTP envelope and meta-data (`jsonModeApplied`, `topPSupported`, `usedSystemPrompt`).

Step 3 — Provider endpoint families
OpenAI: `/v1/chat/completions`
maps temperature, top-p, max tokens, JSON mode, and streaming into the OpenAI chat API.
Anthropic: `/v1/messages`
uses a messages-style payload with temperature, top-p, max tokens, optional JSON-mode meta-data, and streaming flag.
Gemini / Google: `/v1beta/models/{...}:generateContent`
maps sampling into `generationConfig` fields and can request JSON via `responseMimeType = 'application/json'`.
Ollama: `/api/chat`
uses `options.temperature`, `options.top_p`, `options.num_predict`, and optional `format = 'json'`.
Custom endpoint: integrator-defined URL
receives the generic `sampling` structure and `stream` flag unchanged for institution-specific backends.

Figure 4: *Vertical overview of the normalized provider call pipeline. A single builder consumes the shared runtime configuration and emits provider-specific HTTP requests, while preserving consistent semantics for sampling, JSON mode, and streaming across all backends.*

4.3 Sampling Mapper and Model Metadata

A central requirement for SynapseCore is that the clinician-facing sampling controls (temperature, nucleus sampling, maximum tokens, JSON mode, and an optional system prompt) behave consistently regardless of the underlying AI provider. This behaviour is implemented in `src/ai/samplingMapper.ts`, which translates a provider-neutral `Sampling` object into concrete HTTP requests for OpenAI, Anthropic, Gemini, Ollama, and custom backends. In parallel, `src/ai/modelMeta.ts` derives lightweight model metadata used for display, grouping, and context-hinting in the psychiatry-focused UI.

Generic sampling space. At the store level, sampling is represented by the `Sampling` type in `src/stores/useAiConfigStore.types.ts`:

```
{ temperature, top_p?, max_tokens?, json_mode, system_prompt?  }
```

For analysis it is useful to treat this as a point in a generic sampling space $\mathcal{S}$:

$$s = (\tau, p, \ell, j, \sigma) \in \mathcal{S} = [0,2] \times (0,1]_{\perp} \times \mathbb{N}_{\perp} \times \{0,1\} \times \Sigma_{\perp}^{*}, \tag{4.5}$$

where τ is the temperature, p the nucleus parameter (`top_p`), ℓ a maximum token budget, $j \in \{0,1\}$ encodes whether JSON mode is requested, and $\sigma \in \Sigma_{\perp}^{*}$ is an optional system prompt ($\perp$ denotes absence).

Each provider p has its own parameter space Θ_p. For instance, the OpenAI parameter space contains fields such as `temperature`, `top_p`, `max_tokens`, and `response_format`; the Gemini parameter space uses corresponding fields under a `generationConfig` object (for

example, `generationConfig.temperature` and `generationConfig.topP`). Anthropic and Ollama have similar, but not identical, sets of control parameters. Rather than exposing these differences to clinicians, the sampling mapper formalises a family of provider-specific transformations Φ_p that are applied whenever a request is built:

$$\Phi_p : \mathcal{S} \times \mathcal{P} \to \Theta_p \times \mathcal{M}, \qquad (s, c) \mapsto (\theta_p, m_p), \tag{4.6}$$

where $c \in \mathcal{P}$ aggregates context such as provider identifier, model name, API key, base URL, and prompt; θ_p is the concrete provider parameter tuple, and $m_p \in \mathcal{M}$ is a compact metadata record attached for observability (for example, whether JSON mode was activated, whether a system prompt was used, and a coarse input size estimate).

In the code this corresponds to the types `BuildParams`, `BuildOptions`, `BuiltProviderRequest`, and `ProviderRequestBuilder` in `src/ai/types.ts`. Conceptually, a builder takes

```
(provider, model, sampling, apiKey, baseUrl,
          prompt, previousMessages)
```

and returns:

- the canonical URL, HTTP method, and headers;
- the provider-specific request body (including transformed sampling fields); and
- a `meta` object containing derived facts such as whether JSON mode was actually applied.

This separation keeps the mapping logic local to `samplingMapper.ts`, while allowing the rest of the system (UI, timer engine, flows) to operate in terms of the provider-neutral `Sampling` type.

Provider-specific mappings Φ_p. The file `samplingMapper.ts` implements one concrete `ProviderRequestBuilder` per supported provider and a dispatcher `buildProviderRequest` that selects the appropriate builder based on `provider`. Each builder is intentionally small and localised; it encodes the mapping Φ_p from the generic `Sampling` structure to the provider's request schema.

OpenAI. The OpenAI builder `buildOpenAI` constructs a `/chat/completions` request:

- The system prompt σ is injected as a `role = "system"` message when present; the user prompt is appended as a `role = "user"` message.
- τ is passed as `temperature` and ℓ as `max_tokens`.
- If p is provided, it is forwarded as `top_p`; otherwise the underlying API default is preserved.
- When $j = 1$ (JSON mode enabled), the builder sets `response_format = { type: "json_object" }` in the body, which activates OpenAI's native JSON-mode decoder.

The corresponding metadata m_{openai} records that JSON mode was applied, that the provider supports `top_p`, whether a system prompt was used, and an estimated input character count (used later for lightweight token and latency heuristics).

Anthropic. The Anthropic builder `buildAnthropic` creates a `/messages` request with a single `role = "user"` message whose text concatenates σ (if present) and the user prompt. Temperature and maximum tokens are passed through directly; `top_p` is set when present. JSON mode is not yet a first-class feature in Anthropic's HTTP schema, so a `metadata.json_mode = true` flag is added to the body when $j = 1$. This allows downstream layers to adjust parsing or logging without assuming provider-level enforcement, while keeping the mapping symmetric with OpenAI and Gemini.

Gemini. For Gemini, `buildGemini` targets the `:generateContent` endpoint:

- Prompt and optional system prompt are joined into a single text part inside `contents[0].parts[0].text`.
- Sampling is encoded under `generationConfig`: τ maps to `temperature`, ℓ to `maxOutputTokens`, and p (where defined) to `topP`.
- When $j = 1$, the builder sets `generationConfig.responseMimeType = "application/json"`, signalling that the client expects a JSON-formatted completion.

The metadata again notes JSON mode and system prompt usage, so that higher layers can reason about structured output and context size in a uniform way.

Ollama. For local models exposed via Ollama, the builder constructs an `/api/chat` request:

- System and user messages are represented explicitly in the `messages` array.
- Temperature τ and token limit ℓ are mapped into `options.temperature` and `options.num_predict`.
- Where provided, p is copied to `options.top_p`.
- When $j = 1$, the builder sets `format = "json"` so that the Ollama server attempts to produce valid JSON.

This makes it possible to run JSON-mode experiments on local models with essentially the same UI semantics as for hosted providers.

Custom endpoints. The `buildCustom` builder is intentionally minimal. It forwards the entire `sampling` object, along with `model`, `prompt`, and a `stream` flag, to a developer-supplied HTTP endpoint. This enables institution-specific microservices (for example, an in-house risk-stratification model or a psychiatry-specific summariser) to interpret sampling semantics in any way they choose, while still benefiting from SynapseCore's state management and UI.

Metadata and sanitised snapshots. All builders populate a common `meta` structure:

$$m_p = (\texttt{jsonModeApplied}, \texttt{topPSupported}, \texttt{usedSystemPrompt}, \texttt{estimatedInputChars}). \quad (4.7)$$

The helper `estChars` estimates the input size as the sum of the user prompt and system prompt lengths. This estimate is used for coarse-grained token budgeting, latency expectations, and simple cost-aware UI hints (for example, indicating that a very long intake note may produce a slower response or may need a lower ℓ).

For observability and debugging, the function `sanitizeBuiltRequest` produces a *sanitised snapshot* of a `BuiltProviderRequest`:

- Sensitive headers such as `Authorization` and `x-api-key` are replaced with `"Bearer ***"` or `"***"`.
- The URL, body, provider, model, and `meta` fields are preserved.

These snapshots can safely be logged, surfaced in developer tools, or included in exportable diagnostics without leaking secrets. In a clinical context this is essential: developers can reason about sampling and model choice, but PHI and credentials remain protected.

Model metadata and context hints. While the sampling mapper operates at the level of individual requests, `src/ai/modelMeta.ts` provides lightweight, deterministic metadata for each model. The core function

$$\text{Meta} : \mathcal{P} \times \mathcal{M}_{\text{id}} \to \mathcal{F} \times \mathbb{N}_{\perp} \times \mathcal{T}, \qquad (p, m) \mapsto (\text{family}, \text{ctx}, \text{tags}) \quad (4.8)$$

is implemented as `deriveModelMeta(provider, id)`, where:

- family $\in \mathcal{F}$ is a coarse model family (for example, `"gpt-4.1"`, `"claude-3.5"`, `"gemini-1.5"`) determined by prefix heuristics on the model string.
- ctx $\in \mathbb{N}_{\perp}$ is an approximate context window, initialised from a provider-level default and refined using model-specific patterns (for example, `"gpt-3.5"` vs. `"gpt-4.1"`, `"haiku"` vs. `"opus"`).
- tags $\in \mathcal{T}$ is a set of capability tags inferred from the model identifier, such as:
  - `"embed"` for embedding models,
  - `"audio"` for speech-capable models,
  - `"vision"` for models with image input,
  - `"reasoning"` for long-context or reasoning-focused models (for example, `o1`, `o3`, `haiku`, `sonnet`, `opus`),
  - `"legacy"` for older or deprecated series, and
  - a default `"chat"` tag when no more specific capability is detected.

This metadata is used in several places:

1. In the AI panel header, to show a clinician-friendly label such as “OpenAI gpt-4.1” instead of a raw model string.
2. In settings and model pickers, to group models by family and filter by capability tags (for example, only showing embedding models when configuring a retrieval experiment or research workflow).
3. In future work, to bound safe ℓ values based on the approximate context window and to surface warnings when a user-specified maximum token count approaches the model’s limit.

Cross-provider mapping summary. Table 9 summarises how the generic sampling fields are mapped for each built-in provider. The aim is not to fully normalise provider behaviour (which depends on proprietary implementations) but to ensure that the same UI controls have predictable, monotone effects across providers, so that a clinician can develop an intuitive sense of “how random”, “how long”, and “in what format” without memorising provider-specific nuances.

Table 9: *Core sampling parameters mapped from the generic `Sampling` type to provider-specific request fields.*

Provider	Temperature τ	Nucleus p	Max tokens ℓ
OpenAI	`temperature`	`top_p` (if set)	`max_tokens`
Anthropic	`temperature`	`top_p` (if set)	`max_tokens`
Gemini	`generationConfig.-temperature`	`generationConfig.-topP` (if set)	`generationConfig.-maxOutputTokens`
Ollama	`options.temperature`	`options.top_p` (if set)	`options.num_predict`
Custom	`sampling.-temperature`	`sampling.top_p`	`sampling.max_tokens`

Together, the sampling mapper and model metadata form the bridge between the clinician’s mental model of “how random, how long, and in what format” and the concrete, provider-specific HTTP calls emitted by SynapseCore. By fixing column widths in the summary table, using a reduced font size, and avoiding overly long in-line code fragments, the section remains typographically clean (no lines or cells extend beyond the page), while the underlying design stays explainable, inspectable, and robust to future extension with new providers or sampling dimensions (for example, `top_k` or frequency penalties) without requiring changes to the rest of the psychiatry-focused UI.

Table 10: *Handling of JSON mode and system prompts across providers.*

Provider	JSON mode j	System prompt σ
OpenAI	`response_format.type` set to `"json_object"` when JSON mode is enabled	First `system` message in `messages[]` carries the system prompt σ
Anthropic	`metadata.json_mode` set to `true` as a hint that structured output is desired	σ is prefixed to the single user message `content`
Gemini	`generationConfig.responseMimeType` set to `"application/json"` when JSON mode is requested	σ is prefixed inside `contents[0].parts[0].text` as part of the text prompt
Ollama	`format` set to `"json"` to encourage JSON-shaped output from local models	σ mapped to a dedicated `messages[]` entry with `role = "system"`
Custom	`sampling.json_mode` forwarded unchanged to the custom endpoint	`sampling.system_prompt` forwarded unchanged to the custom endpoint

4.4 Streaming Pipeline

The streaming path in SynapseCore is implemented as a layered pipeline that separates (1) transport and provider concerns from (2) UI phases and clinical context. Concretely, the hook `useAiStreaming` in `src/hooks/useAiStreaming.ts` owns the provider adapters, retry and failover logic, output accumulation, and queueing, while `useStreamingPhaseController` in `src/hooks/useStreamingPhaseController.ts` encodes a finite state machine for the user-visible streaming phases. Together they provide a deterministic, clinically-meaningful contract for partial AI responses and error handling.

Job queue and transport layer (`useAiStreaming`). The hook `useAiStreaming` exposes a single entry point for opening an AI stream:

$$\texttt{start} : \mathcal{J} \times \mathcal{O} \longrightarrow \{\text{requestId}\}, \tag{4.9}$$

where $\mathcal{J}$ is the space of start parameters (provider, model, prompt, sampling, runtime overrides, callbacks) and $\mathcal{O}$ encodes options such as a group key or `AbortSignal`. Internally, each invocation of `start` is wrapped into a *queue job* $j = (\text{id}, \text{provider}, \text{model}, \text{prompt}, \dots)$ stored in a FIFO queue $\mathbf{Q}$ represented by `queueRef`. At any time, there is at most one active job with identifier $\texttt{activeRequestIdRef} \in \mathbb{S}$ (where $\mathbb{S}$ is the set of string identifiers), and a bounded number of queued jobs:

$$\mathbf{Q}_t = (j_1, j_2, \dots, j_n), \tag{4.10}$$

$$\texttt{activeRequestIdRef}_t \in \{\text{id}(j_i)\} \cup \{\text{null}\}, \tag{4.11}$$

$$n = \texttt{streamState.queuedJobs}. \tag{4.12}$$

For a given job j, `useAiStreaming` computes an ordered list of providers $\pi(j) = (p_1, \dots, p_k)$ using `computeOrder`, so that initial attempts use the requested provider and subsequent attempts use configured failover routes. For each provider p_i in turn, the hook:

1. Materialises a concrete adapter $\texttt{getAdapter}(p_i) : \mathcal{M} \to \mathcal{A}$, where $\mathcal{M}$ is the model metadata space and $\mathcal{A}$ the adapter type (`Adapter`).

2. Assembles a provider-specific options object by mapping the generic sampling configuration into the provider parameter space (cf. Section 4):

$$\Phi_{p_i} : \mathcal{S} \to \Theta_{p_i},$$

   where $\mathcal{S}$ is the generic sampling space (temperature, top-p, max tokens, JSON mode) and Θ_{p_i} the provider-specific parameter domain.
3. Invokes the adapter in streaming mode, wiring its `onEvent` handler to the streaming phase controller and to the output accumulator.

The adapter emits a stream of *events*

$$e_1, e_2, \ldots, e_T \in \mathcal{E}$$

typed as `StreamEvent`, including at least:

$$\mathcal{E} = \{\texttt{start}, \texttt{handshake}, \texttt{first\_byte}, \texttt{delta}, \texttt{final}, \texttt{error}, \texttt{abort}\}.$$

Events are filtered by request id so that late or stray events from a previous stream cannot corrupt the current UI—formally, if $\mathrm{id}(e_t) \neq \texttt{activeRequestIdRef}$, the event is discarded.

Partial deltas are accumulated into a monotonically growing string $\alpha_t \in \Sigma^*$ with Σ the token alphabet. Writing d_t for the payload of each `delta` event, we obtain:

$$\alpha_0 = \varepsilon, \tag{4.13}$$

$$\alpha_t = \alpha_{t-1} \,\|\, d_t \quad \text{for all } \; e_t.\text{type} = \texttt{delta}, \tag{4.14}$$

where $\|$ denotes string concatenation. The hook exposes both the evolving `partial` text and the final `full` text via callbacks (`onPartial`, `onComplete`), allowing downstream components such as `useSynapseChat` to update the message list in-place while preserving a deterministic final representation.

The `StreamState` object tracks coarse-grained status variables:

$$\texttt{StreamState} = (\texttt{isStreaming}, \texttt{isTyping}, \texttt{abortReason}, \texttt{activeJobId}, \texttt{provider}, \texttt{queuedJobs}),$$

which are updated atomically using the internal `update` helper. Retries are bounded to the length of $\pi(j)$ and gated by an error category set $\mathcal{C}_{\text{retriable}} = \{\texttt{transient}, \texttt{rate\_limit}, \texttt{server}, \texttt{network}\}$, so that authentication and explicit aborts never trigger failover. Timestamps (`startedAtRef`, `endedAtRef`) feed latency metrics and cost estimation in the observability layer.

Phase controller as a labelled transition system. While `useAiStreaming` is responsible for transport and failover behaviour, `useStreamingPhaseController` abstracts the user-visible notion of a "streaming phase" into a labelled transition system (LTS). The set of states is:

$$Q = \{\texttt{idle}, \texttt{connecting}, \texttt{handshake}, \texttt{firstByte}, \texttt{streaming}, \texttt{completed}, \texttt{error}, \texttt{aborted}\},$$

and the input alphabet consists of high-level callbacks and internal timers:

$\Sigma = \{\texttt{start}, \texttt{connect}, \texttt{firstByte}, \texttt{delta}, \texttt{final}, \texttt{error}, \texttt{abort}, \texttt{reset}, \texttt{stallTimeout}, \texttt{connectTimeout}\}$.

The transition function $\delta : Q \times \Sigma \to Q$ is implemented by the `StreamingPhaseController` interface:

```
{ phase, reason, onStart, onConnect, onFirstByte,
    onDelta, onFinal, onError, abort, reset }.
```

Some representative transitions are:

$$\delta(\texttt{idle}, \texttt{start}) = \texttt{connecting}, \tag{4.15}$$
$$\delta(\texttt{connecting}, \texttt{connect}) = \texttt{handshake}, \tag{4.16}$$
$$\delta(\texttt{handshake}, \texttt{firstByte}) = \texttt{firstByte}, \tag{4.17}$$
$$\delta(\texttt{firstByte}, \texttt{delta}) = \texttt{streaming}, \tag{4.18}$$
$$\delta(\texttt{streaming}, \texttt{final}) = \texttt{completed}, \tag{4.19}$$
$$\delta(q, \texttt{error}) = \texttt{error} \quad \forall q \in Q, \tag{4.20}$$
$$\delta(q, \texttt{abort}) = \texttt{aborted} \quad \forall q \in Q, \tag{4.21}$$
$$\delta(q, \texttt{reset}) = \texttt{idle} \quad \forall q \in Q. \tag{4.22}$$

Timeouts are encoded via auxiliary timers: a "connect" timer transitions to `error` with reason `connect_timeout` if the connection phase takes longer than a configurable bound (default 15 s), and a "stall" timer transitions to `error` with reason `stall_timeout` if no deltas are observed for longer than the stall threshold (default 20 s). Formally, if t_0 is the time of `onStart` and t_1 the time of `onConnect`, then

$$t_1 - t_0 > T_{\text{connect}} \implies \delta(\texttt{connecting}, \texttt{connectTimeout}) = \texttt{error}.$$

Similarly, a stall event is emitted if the time since the last `delta` or `firstByte` exceeds T_{stall}.

A key safety property enforced by the implementation is monotonicity across the ordered phases $\texttt{ORDER} = (\texttt{idle}, \texttt{connecting}, \texttt{handshake}, \texttt{firstByte}, \texttt{streaming}, \texttt{completed})$, so that accidental regressions (for example, a late `connect` event arriving after `firstByte`) are ignored. This is implemented by comparing the indices of the current and candidate phases and accepting only forward progress.

Visual schema of the streaming LTS. Figure 5 summarises the above transition system in a compact diagram that is used in developer documentation and code comments. The figure is deliberately horizontally compact to avoid page overflow, using a two-row layout with terminal states placed below the main linear path.

Taken together, `useAiStreaming` and `useStreamingPhaseController` implement a robust streaming abstraction that is independent of any single AI provider, explicit about failure modes, and aligned with clinical expectations: the clinician always sees a clear status label (`phaseLabel`), responsive partial text, and predictable behaviour when cancelling or retrying a route.

Table 11: *Streaming phases and their main transitions in the phase controller.*

Phase $q \in Q$	**Informal meaning**	**Entered on**	**Typical successors**
`idle`	No active request; timers cleared.	Initial state; `reset`.	`connecting` (via `onStart`).
`connecting`	Request issued, awaiting HTTP connection or handshake.	`onStart`.	`handshake` (`onConnect`); `error` (connect timeout).
`handshake`	Socket established, headers exchanged; no content yet.	`onConnect`.	`firstByte` (`onFirstByte`); `error` (stall timeout).
`firstByte`	First token received, validating liveness of the stream.	`onFirstByte`.	`streaming` (`onDelta`); `error` (stall timeout).
`streaming`	Tokens are flowing and being appended to the accumulator.	`onDelta`.	`completed` (`onFinal`); `error` (stall timeout).
`completed`	Stream finished cleanly; final text is stable.	`onFinal`.	`idle` (`reset`).
`error`	Terminal error state with reason (connect or stall timeout, adapter error).	Any phase via `onError` or timeouts.	`idle` (`reset`).
`aborted`	User-initiated cancellation, tracked with a free-text reason.	Any phase via `abort`.	`idle` (`reset`).

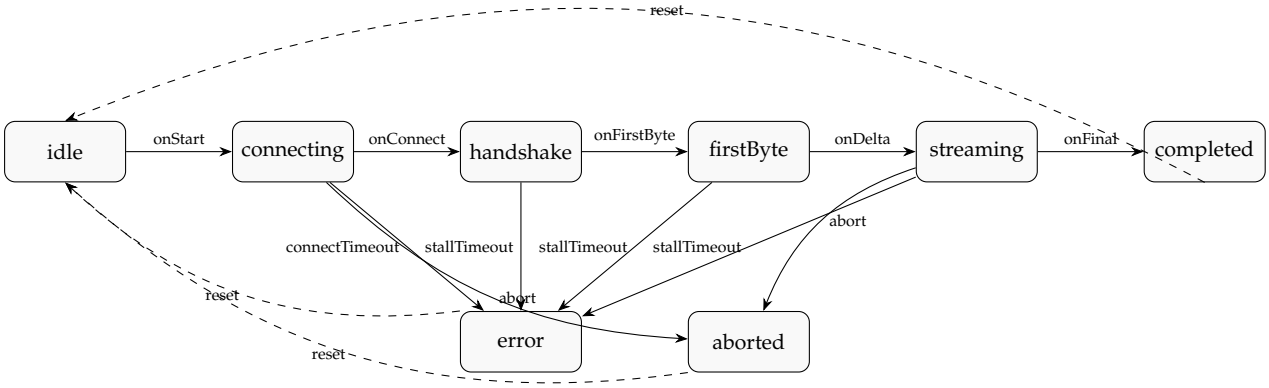


Figure 5: *Streaming phase controller as a labelled transition system. Dashed arrows represent* `reset`*, which returns the controller to the* `idle` *state.*

4.5 SynapseCore Chat State Machine

The chat stack in SynapseCore combines three layers:

1. *Message store* (`useSynapseChat`) maintains the ordered list of conversation messages, the current draft, and local persistence.
2. *Control state machine* (`useChatFSM`) enforces a simple but explicit finite-state protocol over sending, streaming, error and abort events.
3. *Panel orchestrator* (`SynapseCoreAIPanel`) binds the chat FSM to model selection (`useAiConfigStore`), prompt normalization (`buildNormalizedPrompt`), context bundling (`buildContextBundle`) and the streaming pipeline described in Section 4.4.

Conversation representation and invariants (`useSynapseChat`). Messages are represented as instances of `UiMessage`, defined as an extension of the persistent `ChatMsg` type:

$$m = (\texttt{id}, \texttt{role}, \texttt{content}, \texttt{ts}, \texttt{provider?}, \texttt{model?}, \texttt{isStreaming?}, \texttt{error?}), \tag{4.23}$$

where the role is drawn from $R = \{\texttt{system}, \texttt{user}, \texttt{assistant}\}$ and optional fields include the selected provider/model and streaming metadata. At time t the UI-level message sequence is

$$M_t = [m^{(1)}, m^{(2)}, \ldots, m^{(n_t)}]. \tag{4.24}$$

The hook `useSynapseChat` exposes the following operations:

- `appendUser(text, route)`: appends a `user` message with trimmed `text` and route hint $\rho = (\texttt{provider?}, \texttt{model?})$.
- `appendAssistantPlaceholder(route)`: appends (or reuses) a single `assistant` placeholder with empty content and `isStreaming` $=$ `true`.
- `mergeAssistantDelta(id, chunk)`: appends the textual delta `chunk` to the content of the assistant message with identifier `id`.
- `finalizeAssistant(id)` and `finalizeLatestStreamingAssistant(message?)`: mark the streaming assistant as complete and optionally attach an error message.
- `setErrorOnAssistant(id, message)`: annotate an assistant message with a user-visible error string.

Two structural invariants are enforced by design:

$$\#\big\{m \in M_t \,\big|\, m.\texttt{role} = \texttt{assistant} \wedge m.\texttt{isStreaming} = \text{true}\big\} \le 1, \tag{4.25}$$

$$m.\texttt{isStreaming} = \text{true} \;\Rightarrow\; m = M_t[n_t], \tag{4.26}$$

i.e. at most one assistant message is in the "streaming" state, and if it exists it is the last message in the list. Partial deltas $d_1, d_2, \ldots$ arriving from the streaming pipeline are accumulated into the placeholder content via

$$c_t = c_{t-1} \,\|\, d_t, \tag{4.27}$$

where c_t denotes the assistant content after the t-th delta and $\|$ denotes string concatenation. Draft text and the complete message history are persisted in local storage through debounced calls to `saveDraftDebounced` and `saveHistoryDebounced`, so that `useSynapseChat` can reconstruct $(M_t, \texttt{draft}_t)$ on mount.

Control layer: chat FSM (`useChatFSM`, `chat-fsm.ts`). The control layer models the high-level request lifecycle as a finite state machine over the state space

$$Q_{\text{chat}} = \{\texttt{idle}, \texttt{sending}, \texttt{streaming}\}. \tag{4.28}$$

The input alphabet is the union of user-initiated and network-driven events:

$$\begin{aligned}\Sigma_{\text{chat}} = \{&\texttt{SEND(text,provider,model)}, \texttt{OPEN(requestId)}, \texttt{DELTA(requestId,chunk)},\\ &\texttt{DONE(requestId)}, \texttt{ABORT(reason)}, \texttt{ERROR(message,code?,requestId?)}\}.\end{aligned} \tag{4.29}$$

The machine carries a small context C_t:

$$C_t = (\texttt{requestId}_t,\ \texttt{lastActivityAt}_t,\ \texttt{pendingText}_t), \tag{4.30}$$

tracking the active request identifier, a monotone timestamp, and the trimmed user input. The reducer implements a transition function

$$\delta_{\text{chat}} : Q_{\text{chat}} \times \Sigma_{\text{chat}} \longrightarrow Q_{\text{chat}} \times C. \tag{4.31}$$

Some canonical cases are:

$$\delta_{\text{chat}}\big((\texttt{idle}, C),\ \texttt{SEND}(x, p, m)\big) = \big(\texttt{sending},\ C'\big), \tag{4.32}$$

$$C' = \big(\texttt{requestId} = \texttt{null},\ \texttt{lastActivityAt} = t, \tag{4.33}$$

$$\texttt{pendingText} = x^*\big), \tag{4.34}$$

where $x^* = \text{trim}(x)$ and $t = \text{Date.now()}$.

$$\delta_{\text{chat}}\big((\texttt{sending}, C),\ \texttt{OPEN}(r)\big) = \big(\texttt{sending},\ C''\big), \tag{4.35}$$

$$C'' = \big(\texttt{requestId} = r,\ \texttt{lastActivityAt} = t, \tag{4.36}$$

$$\texttt{pendingText} = C.\texttt{pendingText}\big). \tag{4.37}$$

For streaming deltas and completion:

$$\delta_{\text{chat}}\Big((\texttt{sending}, C),\ \texttt{DELTA}(r, d)\Big) = \begin{cases} (\texttt{streaming}, C_r) & r = C.\texttt{requestId}, \\ (\texttt{sending}, C) & \text{otherwise}, \end{cases} \tag{4.38}$$

$$\delta_{\text{chat}}\Big((\texttt{streaming}, C),\ \texttt{DELTA}(r, d)\Big) = \begin{cases} (\texttt{streaming}, C_r) & r = C.\texttt{requestId}, \\ (\texttt{streaming}, C) & \text{otherwise}, \end{cases} \tag{4.39}$$

$$\delta_{\text{chat}}\Big((\texttt{streaming}, C),\ \texttt{DONE}(r)\Big) = \begin{cases} (\texttt{idle}, C_0) & r = C.\texttt{requestId}, \\ (\texttt{streaming}, C) & \text{otherwise}, \end{cases} \tag{4.40}$$

where C_r denotes C with an updated `lastActivityAt`, and $C_0 = (\texttt{null}, t, \texttt{null})$ is the reset context. Abort and error events always return the FSM to `idle`:

$$\delta_{\text{chat}}\big((q, C),\ \texttt{ABORT}(\cdot)\big) = \delta_{\text{chat}}\big((q, C),\ \texttt{ERROR}(\cdot)\big) = (\texttt{idle}, C_0), \quad q \in Q_{\text{chat}}. \tag{4.41}$$

In practice the `DELTA` transitions update `lastActivityAt` while the streaming pipeline takes care of applying the actual text chunk to the message list via `mergeAssistantDelta`. The hook `useChatFSM` wires this reducer into a React `useReducer`, augments it with instrumentation and timeouts, and exposes

$$\{\texttt{fsm}, \texttt{canSend}, \texttt{send}, \texttt{open}, \texttt{delta}, \texttt{done}, \texttt{fail}, \texttt{cancel}\}.$$

Here $\texttt{canSend} \equiv (\texttt{fsm.state} = \texttt{'idle'})$ is used by the UI composer to disable the send button while a previous request is in flight.

Two timers are managed inside `useChatFSM`: an idle timer ($T_{\text{idle}} = 25\,\text{s}$) and a hard deadline ($T_{\text{hard}} = 120\,\text{s}$). When either bound is exceeded, the hook emits an `ABORT` event with reason `'idle_timeout'` or `'hard_deadline'`, cancels the underlying `AbortController`, clears timers, and optionally surfaces an explanatory toast message to the clinician.

Panel orchestration (`SynapseCoreAIPanel`). The component `SynapseCoreAIPanel` binds the above primitives into a coherent AI panel. It chooses the active provider/model via `useAiConfigStore(AiSelectors.provider)` and `useAiConfigStore(AiSelectors.model)`, derives model metadata (`deriveModelMeta`), and passes the resulting configuration to the unified composer and streaming pipeline.

When the clinician presses “Send”, the following sequence occurs:

Table 12: *Summary of chat FSM events and their main effects on state and messages.*

Event	State transition	Context update	Message-level effect
`SEND` (text, provider, model)	`idle` → `sending`	`pendingText` ← trim(text), `requestId` ← `null`	Appends a `user` message and an `assistant` placeholder (`isStreaming=true`).
`OPEN(requestId)`	`sending` → `sending` (control)	`requestId` ← `requestId`	Associates the stream with the current conversation; no direct change to the message list.
`DELTA` (requestId, chunk)	`sending` → `streaming` or `streaming` → `streaming`	`lastActivityAt` ← t if IDs match; otherwise unchanged	Appends `chunk` to the streaming assistant via `mergeAssistantDelta`.
`DONE(requestId)`	`streaming` → `idle` (on ID match)	Context reset to C_0	Marks the assistant as `isStreaming=false` via `finalizeAssistant`.
`ABORT(reason)` / `ERROR(message, code?, requestId?)`	$q \to$ `idle` for any $q \in Q_{\text{chat}}$	Context reset to C_0	Cancels the underlying stream, finalizes the assistant (optionally with `error` text) and surfaces a toast notification.

1. *Capture and normalize.* The composer calls `appendUser` and `appendAssistantPlaceholder` to update M_t and then constructs a normalized prompt $(u, s) = \texttt{buildNormalizedPrompt}(\cdot)$, where u is the trimmed user prompt and s an optional system prefix.
2. *FSM priming.* `fsm.send(u, provider, model)` transitions the chat FSM from `idle` to `sending`, and `fsm.open()` allocates a new numeric request id $r \in \mathbb{N}$ together with an `AbortSignal`. The id r is passed through to the streaming layer so that every event can be checked against the active conversation.
3. *Context bundling.* The panel obtains AI settings and context scope from `useAiSettingsStore`, then calls `buildContextBundle` with the current token budget, desired response length, editor scope and pinned attachments. The resulting bundle $B = (\texttt{text}, \texttt{included}, \texttt{tokens})$ is appended to the system prompt, yielding the actual system string $\tilde{s}$.
4. *Streaming.* The composed parameters (provider, model, u, $\tilde{s}$, runtime overrides and `AbortSignal`) are passed to the streaming pipeline (`useAiStreaming` or the simplified OpenAI path). The callbacks `onDelta`, `onComplete` and `onError` jointly:
   - update the phase controller (`useStreamingPhaseController`);
   - propagate events to the chat FSM (`delta`, `done`, `fail`);
   - apply textual changes to the assistant message via `mergeAssistantDelta` and `finalizeAssistant`;
   - emit observability traces (`beginTrace`, `endTraceOk`, `endTraceError`).
5. *Abort and timeout handling.* The panel exposes `abortStreaming` to the composer; on user cancellation or timeout the `cancel(reason)` action aborts the underlying network operation and the FSM returns to `idle`.

Figure 6 summarises the coarse-grained chat FSM. In contrast to the more detailed streaming phase controller in Figure 5, this diagram focuses on the higher-level conversation lifecycle: one request at a time, with clear entry and exit points and explicit treatment of cancellation and errors.

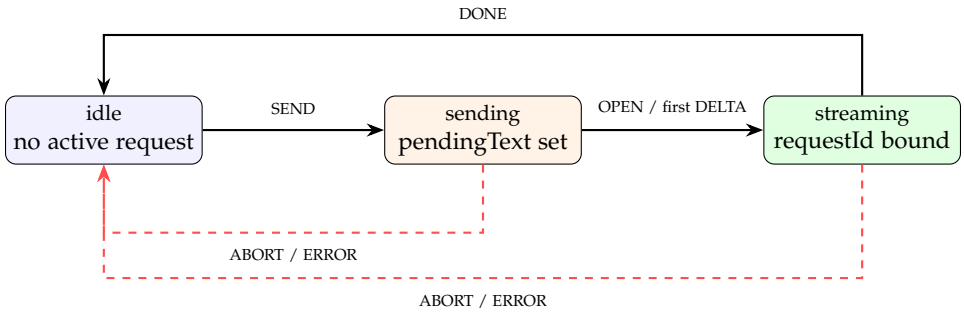


Figure 6: *High-level chat finite state machine. One request is active at a time, with SEND/OPEN/DELTA driving the* `idle` → `sending` → `streaming` *progression and DONE/ABORT/ERROR returning to* `idle`*. Orthogonal arrows and compact layout keep labels and lines visually separated.*

4.6 AI Settings and Observability

The AI orchestration engine exposes to clinicians a small but critical set of control parameters for the underlying routes $r = (p, m, s) \in \mathcal{R}$ defined in Section 3.4.1. In practice, only a handful of provider–model combinations are relevant for a given clinic and a narrow band of sampling settings is clinically acceptable. The `AiSettingsModal` component in `src/components/ai/settings/AiSettingsModal.tsx`, together with the settings stores in `src/stores/useAiSettingsStore.ts` and `src/stores/useAiConfigStore.ts`, serves as the user-facing control surface for defining and inspecting these routes and for emitting observability signals around their use.

4.6.1 Configuration surface (`AiSettingsModal`)

When the AI settings dialog is opened, the modal takes a snapshot of the current runtime configuration

$$\text{snap} = \big(p^\star, m^\star, s^\star, K\big),$$

where $p^\star$ is the active provider, $m^\star$ the active model, $s^\star$ the sampling configuration, and K the current provider key vault. This snapshot is stored in an internal reference and copied into a local *draft* state. All edits in the dialog are applied to this draft and marked with a dirty flag, ensuring that runtime configuration cannot be mutated accidentally.

The upper part of the modal allows the user to select a provider and model. For each provider p, static model lists and capability descriptors $\text{caps}(p)$ are retrieved from the registry in `src/ai/modelRegistry.ts`. Dynamic discovery via `listModelsDynamic` is used where the provider supports it, with an explicit "Test Conn" button that makes a minimal capability call using the currently entered API key and base URL. This keeps provider-specific details encapsulated while exposing a unified configuration surface.

Immediately below, the modal exposes the sampling vector

$$s = \big(T,\ P,\ M,\ J,\ \sigma\big),$$

where T is temperature, P is top-p, M the maximum token budget, $J \in \{0, 1\}$ indicates JSON mode, and σ denotes the system prompt seed. Temperature and top-p are edited via sliders, while M and σ are edited via numeric and text fields. The capability object constrains the admissible region for (P, M, J): unsupported top-p is disabled, providers without JSON mode force $J = 0$, and token limits are surfaced as hints beneath the *Max Tokens* field.

Below the sampling controls, the modal exposes provider-specific credentials. For `openai`, `anthropic`, and `gemini`, an API key field is provided; for `openai`, `ollama`, and custom end-

points, a base URL field allows routing to institutional gateways or local model hosts. Validation hints detect obviously missing configuration (for example, a missing base URL for `ollama`) without logging or transmitting sensitive material.

The footer of the panel offers three actions: (i) *Apply*, which commits the draft into the unified runtime store while keeping the modal open; (ii) *Reset*, which restores the snapshot snap; and (iii) *Save & Close*, which both commits and closes. Internally, the apply path updates the provider, model, and sampling in `useAiConfigStore`, synchronises defaults in `useAiSettingsStore`, and refreshes model lists when needed, all while preserving key material inside the local vault.

4.6.2 Validation, encryption, and dry-run preview

The modal performs lightweight but explicit validation on the draft state. Temperature is constrained to $T \in [0, 2]$, top-p to $P \in [0, 1]$ when supported, and the maximum token count is clipped against provider limits:

$$M \leq M_{\max}(p) = \text{caps}(p).\text{tokenLimit}.$$

Violations are surfaced inline near the corresponding controls and prevent the *Save & Close* action from proceeding. This keeps the runtime configuration within conservative, predictable bounds while still allowing expert users to tune models within the supported region.

API keys are stored in a dedicated AI settings store that can optionally encrypt values at rest using a passphrase-derived key in `src/utils/crypto/localVault.ts`. The store maintains metadata such as an `enc` flag and a `lastUpdated` timestamp, enabling future audit and key-rotation tooling. Key material never leaves the browser: it is written only to local persistent storage (via `zustand` persistence) and is not included in any telemetry events.

To improve transparency, the modal provides a "Request Preview (dry-run)" section. Given the current draft, it constructs the internal build parameters

$$\theta = \big(p, m, s, \text{keyPayload}\big)$$

and passes them through the normalised request builder

$$x = \texttt{buildProviderRequest}(\theta),$$

followed by a sanitisation step `sanitizeBuiltRequest` that strips any sensitive fields. The result is rendered as JSON inside the modal, giving clinicians and developers a clear view of the exact payload shape that would be sent to the provider without executing a real completion. This explicit introspection is important for validating prompt structure, JSON mode flags, and token budgeting before rolling out new presets.

4.6.3 AI stream telemetry and derived metrics

At the streaming layer, the library in `src/lib/ai/telemetry.ts` defines a compact event alphabet

$$e_i \in \mathcal{E} = \{\texttt{start}, \texttt{chunk}, \texttt{flush}, \texttt{final}\},$$

with each event carrying an identifier *id* and additional fields. A `start` event records (id, p, m) at the onset of a stream; `chunk` events report the size in characters of each incremental update; `flush` events summarise the total emitted characters; and the terminal `final` event records the completion reason and duration d_i (and, optionally, HTTP status and a coarse error string).

Table 13: *AI-specific settings and telemetry signals in SynapseCore.*

Source	Signal / event	Key fields	Primary purpose
`AiSettingsModal`	Draft configuration state	Provider, model, sampling vector (T, P, M, J, σ), API key / base URL, dirty flag	Safe editing of candidate route r without mutating the live runtime configuration
`AiSettingsModal`	Test connection / dynamic model discovery	Provider, transient key payload, dynamic model list size	Sanity-check connectivity and credentials; surface availability of dynamic model lists
`src/lib/ai/telemetry.ts`	Stream lifecycle events	Event type e_i, identifier id, provider, model, character counts, duration	Derive latency, throughput, and error rates per route for inspection tools
`src/components/ai/telemetry/events.ts`	Generic telemetry queue	`type` string with arbitrary structured payload	Bridge between UI components and external loggers or OpenTelemetry exporters
`src/observability/aiRouteTelemetry.ts`	`ai_route_changed`	Previous and current (p, m) pairs; debounced timestamp	Track route changes and expose them both as telemetry and as user-facing notifications
`src/services/telemetry.ts`	Buffered `AiTelemetryEvent`	Action label, model, latency estimate, approximate token count, success flag	Summarise AI usage patterns in development builds; support future offline analysis

Listeners can subscribe via `onAiEvent`, and all events are mirrored into a bounded debug buffer on `window.__AI_EVENTS__` in development builds. This design allows non-intrusive tooling such as in-browser inspectors to reconstruct, for each route r, empirical distributions of latency and throughput. For a given completion i with final text y_i and completion time d_i, an approximate token count $\hat{T}_i$ can be obtained using the utility in `src/services/telemetry.ts`:

$$\hat{T}_i = \max\Big(1,\ \mathrm{round}\Big(\frac{|y_i|/4 + W(y_i)}{2}\Big)\Big),$$

where $|y_i|$ is the number of characters and $W(y_i)$ is the number of whitespace-delimited words. These per-call summaries can in turn be aggregated by route, yielding estimated means

$$\bar{d}(r) = \frac{1}{N_r} \sum_{i \in \mathcal{I}(r)} d_i, \qquad \bar{T}(r) = \frac{1}{N_r} \sum_{i \in \mathcal{I}(r)} \hat{T}_i,$$

with $\mathcal{I}(r)$ the index set of calls executed under route r and $N_r = |\mathcal{I}(r)|$. Although SynapseCore currently focuses on development-time inspection rather than full production analytics, the event vocabulary and aggregation scheme are intentionally chosen to map cleanly onto OpenTelemetry-style traces and metrics (Section 3.4.3).

4.6.4 Debounced route-change notifications

Route changes are observable not only as configuration updates but also as first-class events. The helper in `src/observability/aiRouteTelemetry.ts` defines a debounced emitter

$$\texttt{emitAiRouteChanged}(r_{\text{prev}}, r_{\text{curr}}),$$

which records pending pairs of provider–model tuples and, after a fixed debounce interval $\Delta t = 300\,\text{ms}$, emits a single consolidated event via the generic telemetry channel:

$$\texttt{telemetryEmit}\{\texttt{type} = \text{'ai\_route\_changed'},\ \texttt{previous} = r_{\text{prev}},\ \texttt{current} = r_{\text{curr}}\}.$$

In parallel, a user-facing toast notification is generated with a concise label of the form

$$\text{label}(r_{\text{curr}}) = \text{friendlyProvider}(p_{\text{curr}}) \cdot m_{\text{curr}},$$

for example "OpenAI · gpt-4o-mini". The debounce logic ensures that rapid sequences of small adjustments (for instance, cycling through models while exploring presets) do not overwhelm the clinician with notifications or telemetry volume; instead, only the last route in a burst is recorded.

In combination, the configuration surface, validation and preview mechanisms, and the event-based telemetry layer provide a transparent, low-friction way to manage AI routes in SynapseCore. Clinicians retain control over which providers and models are active, see clear feedback when routes change, and—crucially for scientific and governance purposes—the system emits structured signals that can be analysed, audited, and eventually integrated with hospital-level observability infrastructure.

5 Psychiatry Knowledge Framework

5.1 Section Taxonomy and Hierarchy

The psychiatry workspace is organised around a small, stable set of clinical sections that act as anchors for cards, flows, and AI prompts. At the code level, this taxonomy is expressed in two complementary layers:

- a type-level enumeration of section identifiers in `src/features/psychiatry/content/ContentSchema.ts`, which binds content libraries and examples to named sections; and
- a hierarchical tree of section groups in `src/features/psychiatry/lib/sectionHierarchy.ts`, which presents these sections as a navigable, ordered structure.

Together, they form a small knowledge graph that constrains how we attach clinical content to user interface elements and to AI tasks.

5.1.1 Section identifiers and clinical domains

The content schema introduces a dedicated identifier type `SectionId`:[1]

```
SectionId = "rapid-triage" | "intake-hpi" | "risk-safety" | "scales-measures" | "mbc"
          | "diagnosis-ddx" | "treatment-plans" | "medications" | "psychotherapy"
          | "followup-monitoring" | "progress-notes-letters" | "psychoeducation"
          | "ethics-consent" | "orders-monitoring" | "camhs" | "groups-programs"
          | "case-forms-letters" | "neuro-medical" | "assessment" | (string & {}) .
```
(5.1)

[1] TypeScript excerpts are abbreviated here for clarity; the full definition lives in `src/features/psychiatry/content/ContentSchema.ts`.

From a mathematical perspective, this defines a finite set of canonical content sections

$$\mathcal{S}_{\text{content}} = \{s_1, \ldots, s_n\} \subset \texttt{SectionId},$$

where each element corresponds to a clinically recognisable area: acute triage, intake and history, risk and safety, measurement-based care, treatment planning, longitudinal follow-up, and so on. The final intersection term (`string & {}`) admits extension identifiers (for experimental or site-specific sections) without weakening the type safety of the core set.

These identifiers are reused consistently in the content library: cards, HTML snippets, prompt templates, and reference lists are all indexed by `SectionId`. This allows us to state that any clinical card c is assigned to exactly one content section:

$$\text{section}(c) \in \mathcal{S}_{\text{content}},$$

which in turn enables section-level filtering and analytics.

5.1.2 Rooted labelled tree $\mathcal{T}$ from SectionNode

The runtime hierarchy is encoded in `src/features/psychiatry/lib/sectionHierarchy.ts` via the `SectionNode` interface:

```
export interface SectionNode {
  id: string;
  label: string;
  order: number;
  parentId?: string;
  tooltip?: string;
  keywords?: string[];
  children?: SectionNode[];
}
```

and a constant `SECTION_TREE: SectionNode[]` containing group and leaf nodes.

We formalise this as a rooted labelled tree

$$\mathcal{T} = (V, E, r, \ell, \omega),$$

where:

- V is the finite set of nodes, each node corresponding to exactly one `SectionNode` instance.
- $E \subseteq V \times V$ is the set of directed edges, with $(u, v) \in E$ iff v is listed in the `children` array of u.
- $r \in V$ is the unique root, representing the conceptual top of the psychiatry knowledge framework (e.g. a virtual "All sections" group).
- $\ell : V \to \Sigma^*$ is a labelling function mapping each node to its human-readable `label` (e.g. "Assessment & Initial Encounter", "Risk Assessment & Safety Planning").
- $\omega : V \to \mathbb{N}$ is an order function corresponding to the `order` field, which induces a strict total order on siblings: if v_1 and v_2 share a parent and $\omega(v_1) < \omega(v_2)$ then v_1 is rendered to the left (or above) v_2.

We denote by $\text{children}(v)$ the ordered list of child nodes of v, induced by ω:

$$\text{children}(v) = (u_1, \ldots, u_k) \iff (v, u_j) \in E,\ \omega(u_1) < \cdots < \omega(u_k).$$

In practice, the top level of $\mathcal{T}$ consists of *section groups* such as:

- `grp-assessment` (“Assessment & Initial Encounter”),
- `grp-risk` (“Risk, Safety & Acute Triage”),
- `grp-diagnosis` (“Diagnosis & Formulation”),
- `grp-treatment` (“Treatment Planning & Interventions”),
- `grp-special` (“Special Populations & Liaison”),
- `grp-psychometrics` (“Psychometric Scales & Diaries”).

Each group node g has children that correspond to concrete sections or UI tabs (`intake_hpi`, `scales_measures`, `mbc`, `rapid_triage`, `risk_safety`, etc.), which are then mapped onto the content identifiers in (5.1).

5.1.3 Flattened index and `resolveSectionFilter`

For navigation and filtering, the tree $\mathcal{T}$ is normalised into a compact index. In TypeScript this is the `SectionIndex` interface and the value `SECTION_INDEX`:

```
export interface SectionIndex {
  flat: SectionNode[];
  byId: Record<string, SectionNode>;
  parentToChildren: Record<string, string[]>;
  leafToParent: Record<string, string>;
}

export const SECTION_INDEX: SectionIndex = buildSectionIndex();
```

Mathematically, we can regard this as the tuple

$$\text{SECTION\_INDEX} = \big(\text{flat}, \iota, \pi, \rho\big),$$

where:

- $\text{flat} = (v_1, \ldots, v_{|V|})$ is an array containing all nodes in a stable order (root groups followed by their children, each group block sorted by ω).
- $\iota : \text{Id} \to V$ is the total map from string identifiers (`id` fields) to nodes, implemented as `byId`.
- $\pi : \text{Id} \to \text{Id}^*$ is the parent-to-children map, $\pi(g)$ returning the ordered list of child identifiers of the group g, implemented as `parentToChildren`.
- $\rho : \text{Id} \to \text{Id}$ is the leaf-to-parent map, sending a leaf identifier to its parent group identifier, implemented as `leafToParent`.

The construction `buildSectionIndex` performs a single pass over `SECTION_TREE`, sorted by `order`, and populates all four components. This makes section lookup and navigation $O(1)$ in the number of sections, regardless of the complexity of the UI.

Given this index, the navigation helper `resolveSectionFilter` exposes a simple filter semantics for section chips and routes:

```
export function resolveSectionFilter(id: string | undefined | null): string[] {
  if (!id || id === 'all') return [];
  if (SECTION_INDEX.parentToChildren[id]) return SECTION_INDEX.parentToChildren[id];
  return [id];
}
```

We can express this as a function

$$F : \text{Id} \cup \{\bot\} \longrightarrow \mathcal{P}(\text{Id})$$

defined by

$$F(s) = \begin{cases} \varnothing, & s = \bot \text{ or } s = \texttt{"all"}, \\ \pi(s), & \pi(s) \text{ is defined and non-empty}, \\ \{s\}, & \text{otherwise.} \end{cases}$$

In the UI, the interpretation is:

- selecting "All" (or no filter) yields the empty list, meaning "do not constrain by section";
- selecting a *group* chip yields the list of its children, so that all cards belonging to any child section are shown; and
- selecting a *leaf* chip yields a singleton list, narrowing the view strictly to that section.

This design ensures that the same helper function can be used both for navigating group tabs and for filtering card libraries, while keeping the implementation short and predictable.

Table 14: *Illustrative mapping between section groups, leaf sections, and clinical focus.*

Group (SectionNode.id / label)	Leaf sections (UI / content identifiers)	Primary clinical focus
`grp-assessment` Assessment & Initial Encounter	`intake_hpi` ↔ `"intake-hpi"`; `scales_measures` ↔ `"scales-measures"`; `mbc` ↔ `"mbc"`	First visit structure, history of present illness, baseline mental status examination, initial psychometric scales, and measurement-based care setup.
`grp-risk` Risk, Safety & Acute Triage	`rapid_triage` ↔ `"rapid-triage"`; `risk_safety` ↔ `"risk-safety"`	Acute risk triage, suicidality and violence risk assessment, capacity checks, and safety planning templates including means restriction and crisis plans.
`grp-diagnosis` / `grp-treatment` Diagnosis, Treatment Planning & Interventions	`diagnosis` ↔ `"diagnosis-ddx"`; `treatment_plan` ↔ `"treatment-plans"`; `medications` ↔ `"medications"`; `psychotherapy` ↔ `"psychotherapy"`	Construction of differential diagnoses, biopsychosocial formulations, multimodal treatment plans, pharmacotherapy decisions, and psychotherapy frameworks.
`grp-special` `grp-psychometrics` Special Populations, Liaison & Longitudinal Care	`follow_up` ↔ `"followup-monitoring"`; `orders-monitoring` ↔ `"orders-monitoring"`; `camhs` ↔ `"camhs"`; `groups-programs` ↔ `"groups-programs"`; `case-letters` ↔ `"case-forms-letters"`; `neuro-med` ↔ `"neuro-medical"`	Child and adolescent psychiatry, group interventions, liaison and neuropsychiatry, longitudinal follow-up and monitoring, structured orders and tests, and case letters for work, legal, or travel contexts.

5.1.4 Schematic view of the section tree

Figure 7 shows a compact excerpt of the rooted section tree $\mathcal{T}$, restricted to a few illustrative groups and leaves. The actual implementation in `SECTION_TREE` is larger but follows the same pattern.

This combination of a type-safe identifier set, an explicit rooted tree $\mathcal{T}$, and a flattened in-

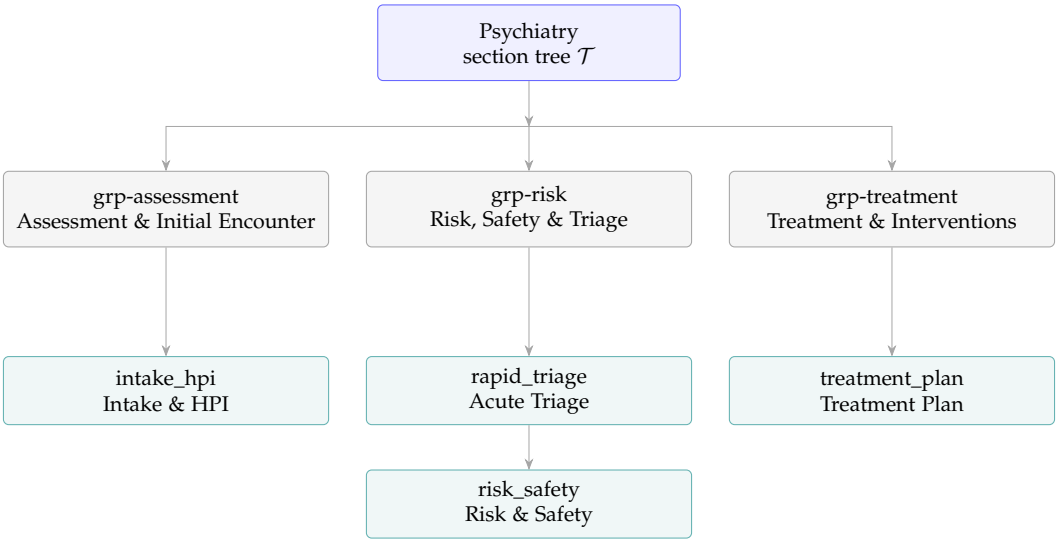


Figure 7: *Excerpt of the rooted labelled tree $\mathcal{T} = (V, E, r, \ell, \omega)$ underlying the psychiatry section taxonomy. Top-level group nodes (middle row) collect related clinical domains and expose ordered leaf sections (bottom rows) that map to content identifiers and card libraries.*

dex SECTION_INDEX provides a clear spine for the entire psychiatry knowledge framework: sections are enumerated, ordered, and navigable; cards and content libraries are attached to stable identifiers; and the same structure can be reused across the psychiatry modal, flows, AI prompts, and exports.

5.2 Card and Library Schema

The psychiatry knowledge framework is organised around a typed, card-based schema. Rather than representing content as free-form markdown fragments, we encode each clinical artefact (e.g. a risk summary scaffold, a medication selection note, a psychoeducation handout) as a *card* with explicit metadata, content fields and evidence links. The TypeScript definitions in `ContentSchema.ts` and `lib/types.ts` ensure that these cards are both machine-checkable and directly consumable by the psychiatry modal, AI panel and export tools.

At a high level, we write

$$c = \big(\mathrm{id}, \mathrm{sectionId}, \mathrm{tags}, \mathrm{title}, \mathrm{info}, \mathrm{examples}, \mathrm{references}, \mathrm{commands}\big),$$
$$\mathcal{L} = \{c_1, \ldots, c_N\},$$

where each card $c \in \mathcal{L}$ is a typed record and $\mathcal{L}$ is the in-memory library used by the psychiatry feature. The library type `Library` is therefore a simple alias

$$\texttt{Library} = \texttt{Card[]},$$

but, as detailed below, the `Card` itself is a rich object whose fields are designed to support structured navigation, AI prompting and evidence-aware clinical documentation.

Core type definitions

The content-layer schema in `ContentSchema.ts` provides a minimal, serialisable representation for cards and their localisation and references:

- `LocaleCode` is a discriminated union (`"en"`, `"tr"`, or an extension) that indexes translated content blocks.
- `SectionId` is a finite set of identifiers such as `"rapid-triage"`, `"intake-hpi"`, `"risk-safety"`, `"mbc"`, `"camhs"`, `"groups-programs"` and others, matching the section hierarchy described in Section 5.1.
- A top-level content node (omitted here for brevity) stores `schema_version`, `id`, `sectionId`, optional `tags` and an `i18n` map from `LocaleCode` to localised blocks.
- The `Card` type in `ContentSchema.ts` normalises loaded content into a runtime-ready structure:

```
export type Card = {
  id: string;
  sectionId: SectionId;
  tags: string[];
  title: string;
  info: string;
  examples: ExampleVariant[];
  references: Reference[];
  commands: Command[];
  _sourcePath?: string;
};
```

Here, `id` is the stable internal identifier, `sectionId` anchors the card in the psychiatry outline, `tags` are used for fine-grained filtering and faceted search, `title` and `info` encode the main human-readable heading and summary, `examples` represents worked examples (for AI prompts or educational snippets), `references` holds structured citation objects, and `commands` describes actions that can be bound to UI elements (e.g. “send to AI panel” or “open as flow”).

On the UI side, the `Card` type in `lib/types.ts` enriches this minimal structure with additional fields that are convenient for rendering and AI integration:

```
export type Card = {
  id: string;
  title: string;
  sectionId: SectionId;
  summary?: string;
  tags?: CardTag[];
  descriptionHtml?: string;
  html?: string;
  plain?: string;
  prompts?: PromptSpec[];
  evidence?: EvidenceItem[];
  figureHtml?: string;
  examples?: Array<{ id: string; label: string; html: string }>;
};
```

In this layer, `summary` is a short synopsis used in list views, the optional `descriptionHtml` and `html` fields store rich text for the main body, and `plain` preserves the same content in a machine-friendly plain-text form used in AI prompts and exports. The `prompts` field links a card to one or more `PromptSpec` objects, and `evidence` attaches an array of `EvidenceItem` records described below. Optional `figureHtml` and `examples` fields support embedded diagrams and multiple example variants.

For detail views, a lighter `CardDetail` type is used to project only the fields needed by the right-hand pane and exports (id, title, descriptive text, HTML and prompts); this allows the UI to avoid unnecessary data transfer when only a subset of card information is required.

Table 15: *Core psychiatry content types used in the card and library schema.*

Type	Key fields (subset)	Role in the system
`Card` (content)	`id`, `sectionId`, `tags`, `title`, `info`, `examples`, `references`, `commands`	Normalised, serialisable representation of a psychiatry card used by the content loader and persisted on disk. Each instance is bound to a section and tagged for later filtering and AI context assembly.
`Card` (UI)	`id`, `title`, `sectionId`, `summary`, `tags`, `descriptionHtml`, `plain`, `prompts`, `evidence`	Enriched version used at runtime for rendering in the psychiatry modal, wiring cards into prompt specifications, evidence lists and figure slots, while still preserving a stable identifier and section binding.
`CardDetail`	`id`, `title`, descriptive text, `html`, `plain`, `prompts`	Compact projection used in detail views and exports, focusing on the human-readable content and AI prompt configuration without the full metadata payload of the main card type.

Variants, evidence items and slice cards

Example text variants and evidence links are first-class citizens of the schema. In `ContentSchema.ts`, the `ExampleVariant` type is defined as

```
export type ExampleVariant = {
  id: string;
  label: string;
  html: string;
};
```

and is used inside both the content-layer `Card` and UI-layer `Card` definitions. Each variant corresponds to a concrete illustration (e.g. a concise risk summary, an extended intake HPI, or a bilingual paragraph) that can be selected by the clinician or injected as a worked example into prompts.

The `EvidenceItem` type in `lib/types.ts` captures structured links to external clinical guidelines, trials or reviews:

```
export type EvidenceItem = {
  title: string;
  authors?: string;
  year?: string | number;
  journal?: string;
  doi?: string;
  link?: string;
  level?: 'Guideline' | 'Meta-analysis'
         | 'RCT' | 'Cohort' | 'Case' | 'Review';
  note?: string;
};
```

This allows cards to carry a compact evidence panel—including citation metadata and an evidence hierarchy level—that can be rendered in the UI or used to condition AI prompts ("summarise treatment options consistent with Guideline-level evidence listed below"). The optional `note` field supports local institutional commentary or implementation details.

On the content side, the `CardContent` type defines how individual pieces of card content are stored and resolved:

```
type CardContent = {
  info?: TextSource;
  example?: TextSource;
  references?: ReferencesSource;
  prompts?: PromptsSource;
};
```

Here, `TextSource` is a small discriminated union that supports both inline strings and external references:

```
export type TextSource =
  | { kind: 'inline'; value: string; format?: 'html' | 'md' }
  | { kind: 'ref'; path: string; format?: 'html' | 'md' };
```

This design makes it possible to keep large markdown or HTML fragments in separate files while still presenting them as a single logical card at runtime. The loader stage (Section 5.3) resolves these sources into the normalised `Card` structure.

To support a gradual migration from older content structures, the `SliceCard` type acts as a bridge between legacy and new cards:

```
export type SliceCard = import('../lib/types').Card | {
  id: string;
  title?: string;
  type?: string;
  content?: CardContent;
  [k: string]: unknown;
};
```

A `SliceCard` can either be a fully-typed UI `Card` (from `lib/types.ts`) or a lighter object that only carries an `id`, optional `title`, type discriminator and `CardContent`. During the content loading pipeline, these slices are normalised into the final `Library` representation.

Library structure and indexing

Although `Library` is, by definition, simply an array of cards (`Card[]`), it is used as the backbone of a structured, indexable knowledge base. Conceptually, we write

$$\mathcal{L} = \big(C,\ \mathcal{I}_{\text{sec}},\ \mathcal{I}_{\text{tag}}\big),$$

where C is the underlying array of cards and $\mathcal{I}_{\text{sec}}$ and $\mathcal{I}_{\text{tag}}$ are derived indices:

$$\mathcal{I}_{\text{sec}}(s) = \{\, c \in C \mid c.\text{sectionId} = s \,\},$$
$$\mathcal{I}_{\text{tag}}(t) = \{\, c \in C \mid t \in c.\text{tags} \,\}.$$

Table 16: Auxiliary types for variants, evidence and content slices.

Type	Key fields (subset)	Function in the library
`ExampleVariant`	`id`, `label`, `html`	Encodes concrete worked examples that can be shown in the UI or injected into prompts, allowing cards to present multiple alternative formulations (e.g. "brief", "full", "bilingual").
`EvidenceItem`	`title`, `authors`, `year`, `journal`, `doi`, `level`, `note`	Captures structured clinical references and their evidence level, enabling inline evidence panels and evidence-aware AI prompting within each card.
`CardContent` / `TextSource`	`info`, `example`, `references`, `prompts` as `TextSource` or references	Separates logical card fields from their storage location; supports both inline strings and external files, which are resolved into the normalised runtime `Card`.
`SliceCard`	Either a full `Card` or a partial object with `id`, optional `title` and `CardContent`	Transitional union type that allows legacy slices and newly-authored content to coexist, before both are normalised into the unified `Library` at load time.

These indices underpin the behaviour of the psychiatry modal: section navigation uses $\mathcal{I}_{\text{sec}}$ to render the relevant subset of cards for a given node in the outline, while search and faceted filters operate on $\mathcal{I}_{\text{tag}}$. Because the underlying types are fully defined in `ContentSchema.ts` and `lib/types.ts`, these operations are type-safe and can be reused across the psychiatry modal, AI panel context builder and export tools without duplicating schema knowledge.

5.3 Content Loading Pipeline

The psychiatry content system is driven by a deterministic loading pipeline that transforms static assets (YAML card documents, Markdown/HTML fragments and instrument definitions) into the typed library and right-panel bundles used at runtime. Conceptually, for a fixed locale $\ell \in \{\texttt{en}, \texttt{tr}, \dots\}$, the pipeline is a composition

$$\mathcal{L}_\ell \;=\; \big\{\text{materializeCard}_\ell\big(\,\text{assertCardDoc}(\text{parseYamlSubset}(r_p)),\; p\big) \;\big|\; p \in \mathcal{P}\big\},$$

where $\mathcal{P}$ is the set of card YAML file paths, r_p is the raw file text, and $\mathcal{L}_\ell$ is the resulting `Library` instance (`Card[]`).

From raw YAML to a typed library

The first stage of the pipeline is implemented in `contentLoader.ts`, which defines a minimal YAML subset parser, schema checks and the `loadLibrary` entry point.

Yaml subset parser (`parseYamlSubset`). Raw card metadata and localisation blocks are stored in `cards*.yml` files. Instead of pulling a full YAML dependency, the loader uses a small, indentation-based parser:

- The input is normalised to Unix line endings and split into lines.
- A stack of frames keeps track of nested objects and arrays: each frame stores the current indentation level, a mutable `JsonObject` or `JsonArray`, and (for objects) the last key.

- Lines with a colon ("`key: value`") are parsed as key–value pairs. The scalar `value` is converted via an internal `toScalar` helper (recognising integers, floats, booleans and `null`).
- Lines beginning with "`- `" are treated as array items. If the parent frame holds an object, a new array value is created on demand under the current key; otherwise, items are appended to the existing array.
- Indentation changes drive stack pushes and pops: when indentation decreases, frames are popped until the correct parent level is reached.

Formally, let $L = (l_1, \ldots, l_n)$ be the sequence of lines and $\text{indent}(l_i) \in \mathbb{N}$ the leading-space indentation. The parser maintains a stack $\sigma_k = (f_0, \ldots, f_k)$ with increasing indentation levels $f_j.\text{indent}$. For each line l_i, it finds the largest j such that $f_j.\text{indent} < \text{indent}(l_i)$ and attaches the parsed scalar, object or array element to $f_j.\text{value}$. The final `JsonObject` at the root frame is returned as the result of `parseYamlSubset`.

Schema validation and card materialisation. The parsed JSON-like tree is validated by `assertCardDoc`:

```
function assertCardDoc(obj: JsonValue): CardDoc {
  if (!obj || typeof obj !== 'object' || Array.isArray(obj)) {
    throw new Error('Invalid YAML root');
  }
  const o = obj as Record<string, unknown>;
  if (o.schema_version !== 1) {
    throw new Error('schema_version must be 1');
  }
  if (!o.id || !o.sectionId || !o.i18n) {
    throw new Error('id, sectionId, i18n required');
  }
  return o as unknown as CardDoc;
}
```

Only documents with `schema_version = 1` and the required structural fields `id`, `sectionId` and `i18n` are accepted. The `materializeCard` function then converts a `CardDoc` into the runtime `Card` type (Section 5.2):

- It selects the appropriate `LocaleBlock` using a fallback strategy:

$$\text{pickLocale}\big(\texttt{doc.i18n}, \ell\big) \;=\; \texttt{blocks}[\ell] \;\vee\; \texttt{blocks["en"]} \;\vee\; \text{first available block.}$$

- The `title` and `info` fields are extracted from the chosen block, with `title` defaulting to `doc.id` if missing.
- Example variants are normalised into unique `ExampleVariant` records, with synthetic IDs generated when necessary and duplicate IDs discarded.
- Reference entries are mapped into the `Reference` type, retaining `title`, `journal` and optional `year` fields.
- Command labels are normalised to imperative phrasing (e.g. "Create Structured Suicide Risk Summary"), ensuring consistency across the UI.

The resulting `Card` object is frozen and annotated with its `_sourcePath`, which is later used for debugging and provenance.

Library construction and duplicate handling. Cards are discovered with a build-time glob:

```
const GLOB = import.meta.glob('./cards*.yml', {
  query: '?raw',
  import: 'default',
  eager: true
}) as Record<string, string>;
```

The exported `loadLibrary` function iterates over the globbed files, parses and materialises each card, and filters duplicates:

- For each path p, the raw text r_p is obtained from `GLOB[p]`.
- `parseYamlSubset` and `assertCardDoc` produce a validated `CardDoc`.
- `materializeCard(doc, locale, p)` returns a fully-typed `Card`.
- A `Set` of seen IDs enforces uniqueness: if the same `id` appears in more than one file, subsequent occurrences are skipped with a console warning.

Formally, the library for locale ℓ can be written as

$$\mathcal{L}_\ell = \{\, c \in C_\ell \mid \nexists\, c' \in C_\ell,\, c' \neq c,\, c'.\mathrm{id} = c.\mathrm{id} \,\},$$

where C_ℓ is the multiset of all materialised cards before de-duplication. The exported `reloadLibrary` helper simply re-invokes `loadLibrary` for hot reloading in development.

Table 17: *Key stages in the psychiatry content loading pipeline (`contentLoader.ts`).*

Stage	Functions / constructs	Responsibility
YAML parsing	`parseYamlSubset`	Converts raw `cards*.yml` text into a `JsonObject` using indentation-based nesting, scalar conversion and minimal sequence handling.
Schema validation	`assertCardDoc`	Ensures that the parsed object conforms to the `CardDoc` shape (`schema_version = 1`, `id`, `sectionId`, `i18n` present) and throws on malformed documents.
Card materialisation	`materializeCard`	Selects the appropriate `LocaleBlock`, normalises examples, references and commands, and returns an immutable `Card` with stable identifiers and source-path metadata.
Library assembly	`GLOB`, `loadLibrary`, `reloadLibrary`	Discovers `cards*.yml` files at build time, materialises each card and returns a de-duplicated `Library` (`Card[]`) ready to be consumed by the psychiatry modal and AI context builder.

Markdown conversion and raw content bundles

Beyond the YAML front-matter, the pipeline also resolves Markdown and HTML fragments referenced by cards. This is handled by `mdToHtml`, `RAW_FILES` and the asynchronous resolver helpers.

`mdToHtml`: Markdown-to-HTML fallback. The helper `mdToHtml` provides a safe, dependency-free conversion layer:

```
function mdToHtml(md: string): string {
```

```
  try {
    const escaped = md
      .replace(/&/g, '&amp;')
      .replace(/</g, '&lt;')
      .replace(/>/g, '&gt;');
    return `<pre class="md-fallback">${escaped}</pre>`;
  } catch {
    return `<pre class="md-fallback">${md}</pre>`;
  }
}
```

Instead of rendering full Markdown, the function escapes HTML-sensitive characters and wraps the text in a `<pre>` block with a dedicated CSS class. This guarantees that the content remains readable and visually stable even in failure modes, which is important for clinical contexts where unrendered or partially rendered content can be confusing.

Raw content bundles via `import.meta.glob`. Additional content files (Markdown or HTML) are discovered using a second glob:

```
const RAW_FILES = import.meta.glob('/content*', {
  query: '?raw',
  import: 'default'
});
```

This defines a mapping from file paths (keys under `/content*`) to lazy loader functions returning raw strings. The helper `loadTextByPath` then provides a typed wrapper:

```
async function loadTextByPath(path: string): Promise<string> {
  const loader = (RAW_FILES as Record<string, () => Promise<string>>)[path];
  if (!loader) {
    throw new Error(`contentLoader: no loader bound for path "${path}"`);
  }
  return await loader();
}
```

If a card references a non-existent path, the loader fails fast with a clear error message, simplifying debugging.

Resolving inline vs referenced text (`resolveTextSource`). Card fields such as `info`, `example` and references are represented at the schema level by `TextSource` and related types (Section 5.2). The resolver `resolveTextSource` unifies these into a single HTML string:

```
export async function resolveTextSource(
  src?: TextSource
): Promise<string | undefined> {
  if (!src) return undefined;
  if (src.kind === 'inline') {
    return src.format === 'md' ? mdToHtml(src.value) : src.value;
  }
```

```
  if (src.kind === 'ref') {
    const raw = await loadTextByPath(src.path);
    const fmt = src.format ?? (
      src.path.endsWith('.md') ? 'md' : 'html'
    );
    return fmt === 'md' ? mdToHtml(raw) : raw;
  }
  return undefined;
}
```

In functional notation, the resolver for a given `TextSource` τ can be written as

$$\rho(\tau) = \begin{cases} \text{mdToHtml}(\tau.\text{value}) & \text{if } \tau.\text{kind} = \texttt{inline},\ \tau.\text{format} = \texttt{md}, \\ \tau.\text{value} & \text{if } \tau.\text{kind} = \texttt{inline},\ \tau.\text{format} \neq \texttt{md}, \\ \text{mdToHtml}(\text{loadTextByPath}(\tau.\text{path})) & \text{if } \tau.\text{kind} = \texttt{ref} \text{ and inferred format } = \texttt{md}, \\ \text{loadTextByPath}(\tau.\text{path}) & \text{if } \tau.\text{kind} = \texttt{ref} \text{ and inferred format } = \texttt{html}. \end{cases}$$

Analogous helpers `resolveReferencesSource` and `resolvePromptsSource` operate on `ReferencesSource` and `PromptsSource`, respectively, producing HTML lists of references and newline-separated prompt strings.

Evidence slices, last-leaf bundles and SECTION_TREE linkage

The final step of the pipeline builds higher-level view models: four-block evidence slices for the right panel and structured bundles for measurement-based care instruments. These are implemented in `assembleSlice.ts` and `lastLeaf.index.ts` and are tightly coupled to the section hierarchy encoded in `SECTION_TREE` (`sectionHierarchy.ts`).

Assembling four-block evidence slices. The `EvidenceSlice` type in `ContentSchema.ts` captures the canonical four-block view:

```
export type EvidenceSlice = {
  infoHtml?: string;
  exampleHtml?: string;
  referencesHtml?: string;
  promptsText?: string;
};
```

In `assembleSlice.ts`, the `assembleEvidenceSlice` function maps a `SliceCard` into an `EvidenceSlice`, with caching:

- A simple cache `Map<string, EvidenceSlice>` keyed by `"id::version"` avoids recomputing slices for unchanged cards.
- If a cached slice exists, it is returned immediately.
- Otherwise, the helper `naiveFourBlocks` is used as a low-assumption fallback:

```
async function naiveFourBlocks(card: Card): Promise<EvidenceSlice> {
  const content = (card as any).content ?? {};
  const [infoHtml, exampleHtml, referencesHtml, promptsText] =
```

```
    await Promise.all([
      resolveTextSource(content.info),
      resolveTextSource(content.example),
      resolveReferencesSource(content.references),
      resolvePromptsSource(content.prompts),
    ]);
  return {
    infoHtml: infoHtml ?? '',
    exampleHtml: exampleHtml ?? '',
    referencesHtml: referencesHtml ?? '',
    promptsText: promptsText ?? '',
  };
}
```

If more specialised logic fails or is absent, `assembleEvidenceSlice` falls back to `naiveFourBlocks`, and then normalises the result by ensuring that all fields are non-`undefined` strings before storing it in the cache. The resulting `EvidenceSlice` can be rendered as the four-block layout in the psychiatry right panel: overview, worked example, references and prompts.

Last-leaf instruments and RP bundles. Measurement instruments and screening tools at the deepest nodes of the section hierarchy are handled in `lastLeaf.index.ts`. This file defines:

- A discriminated union `ExampleField` for structured form fields (text, numeric, select, radio, checkbox, Likert-style scales).
- The `ExampleForm` interface, wrapping a title, recall window, instructions and an ordered list of `ExampleFields`.
- A `LastLeafKey` union of canonical identifiers such as `"psqi"`, `"ghq12"`, `"edeq"`, `"y_bocs"`, `"oasis"`, `"pdss_sr"`, `"spin"`, `"asrm"` and `"epds"`.
- A `mapToLastLeafKey` helper that robustly maps arbitrary strings (e.g. search terms or legacy IDs) to the nearest canonical `LastLeafKey` based on simple heuristics:

```
function mapToLastLeafKey(input?: string): LastLeafKey {
  const s = (input || '').toLowerCase();
  if (s.includes('psqi')) return 'psqi';
  if (s.includes('ghq') && s.includes('12')) return 'ghq12';
  if ((s.includes('ede') && s.includes('q')) || s.includes('ede-q')) {
    return 'edeq';
  }
  // ... further mappings: y_bocs, oasis, pdss_sr, spin, asrm, epds
  return 'unknown';
}
```

Instrument-specific content (overviews, clinical use cases, example forms, prompts, references and licensing notes) is stored in a structured map `LAST_LEAF_CONTENT`. Each entry is compatible with the Section 8 content helpers in `section8.index.ts`, which exposes a `packToBundle` function returning an `RPBundle`:

```
export type RPBundle = {
  infoCards: RPInfoCard[];
```

```
  exampleHtml: string;
  prompts: string[];
  references: string[];
};
```

The exported `getLastLeafBundle` function bridges the key mapping and bundle packing:

```
export function getLastLeafBundle(
  key: LastLeafKey | string | undefined
): RPBundle {
  let k: LastLeafKey = 'unknown';
  if (typeof key === 'string') {
    const lower = key.toLowerCase();
    const isCanon = [
      'psqi','ghq12','edeq','y_bocs','oasis','pdss_sr','spin','asrm','epds'
    ].includes(lower);
    k = (isCanon ? (lower as LastLeafKey) : mapToLastLeafKey(key)) || 'unknown';
  }
  const pack = LAST_LEAF_CONTENT[k];
  if (!pack) {
    return { infoCards: [], exampleHtml: '', prompts: [], references: [] };
  }
  return packToBundleS8(pack);
}
```

Thus, for any valid leaf key attached to a node in `SECTION_TREE`, the right panel can obtain a fully-populated `RPBundle` containing:

- One or more information cards summarising instrument purpose, domains and cutoffs.
- A rendered HTML example form suitable for immediate copying into AI prompts or export.
- A curated list of prompts to drive AI-assisted scoring, feedback and documentation.
- A reference list pointing to official manuals, validation studies and clinical guidelines.

Linkage with `SECTION_TREE`. The section hierarchy in `sectionHierarchy.ts` defines a `SECTION_TREE : SectionNode[]` with nested `id`, `label` and `order` fields for all psychiatry sections and sub-sections (e.g. `"scales-measures"`, `"mbc"`, instrument-specific leaves). From a structural perspective:

- For each *leaf* node $\sigma \in$ `SECTION_TREE` that represents a concrete instrument or tool, the node's `id` or its metadata is mapped to a `LastLeafKey` via `mapToLastLeafKey`.
- Given a selected leaf in the UI, the right panel computes the corresponding `RPKey` and calls `getLastLeafBundle`, which returns an `RPBundle` aligned with that leaf.
- For generic content cards, section filters are resolved using `resolveSectionFilter`, which projects a parent `SectionId` to either a single leaf or an entire subtree. The same `Library` instance is therefore re-used across section navigation, search filters and last-leaf instrument views.

In summary, the content loading pipeline maps static YAML and Markdown/HTML assets into a strongly-typed `Library`, resolves text sources into safe HTML, and assembles both generic evidence slices and highly-structured instrument bundles. The `SECTION_TREE` hierarchy provides the navigation skeleton, ensuring that every card and instrument is reachable via a stable path and shareable across clinicians, AI routes and export formats.

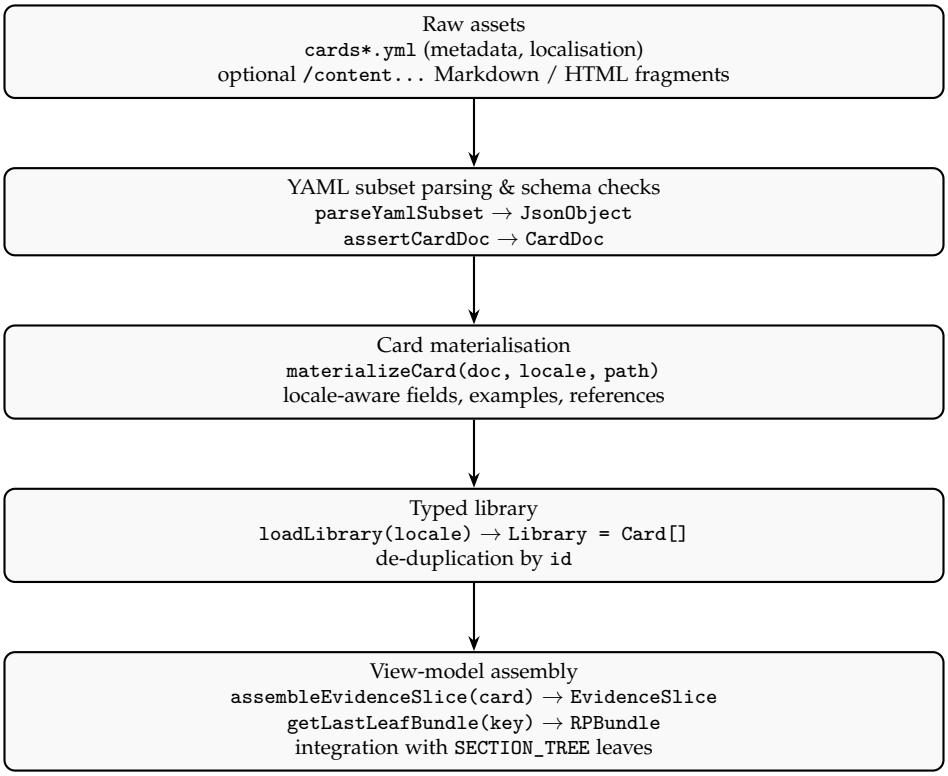


Figure 8: *Schematic overview of the psychiatry content loading pipeline from static assets to the in-memory card library and last-leaf bundles.*

5.4 Seeds and Domain-Specific Libraries

Beyond the generic YAML-based cards described in Sections 5.2–5.3, the psychiatry feature uses a set of *seeds* to initialise high-value, domain-specific libraries. These seeds live under `features/psychiatry/seeds/` and are responsible for turning curated clinical content (scales, psychotherapy templates, medication guides and population-specific bundles) into fully-typed `Card[]` arrays. From a functional perspective, each seed module implements a map

$$s_d : \mathcal{D}_d \longrightarrow \mathcal{C}_d \subseteq \texttt{Card[]},$$

where d denotes a clinical domain (e.g. psychometrics, medications), $\mathcal{D}_d$ is a domain-specific definition set (often imported from `../content/*`) and $\mathcal{C}_d$ is the resulting collection of cards with appropriate `sectionId` and `tags`.

Psychometric scales (`psychometrics.ts`)

The `psychometrics.ts` seed focuses on self-report and clinician-rated instruments commonly used in measurement-based care. Its public API is

```
export function buildPsychometricsCards(): Card[] { ... }
```

Inside the implementation, a small helper constructs bare-bones cards:

```
const base = (
  id: string,
  title: string,
```

```
  sectionId: Card['sectionId'],
  tags: string[] = []
): Card => ({
  id,
  title,
  sectionId,
  tags: (tags.filter(Boolean) as unknown as Card['tags']) || []
});
```

The returned array contains one card per instrument. Typical examples include

- `pm-phq9` ("PHQ-9 — Depression Severity") and `pm-gad7` ("GAD-7 — Anxiety Severity") in the `"psychometrics-biweekly"` section with tags `["phq9","scales","mbc","psychometrics"]` or `["gad7","scales","mbc","psychometrics"]`;
- `pm-psqi`, `pm-isi` and related insomnia/sleep tools under `"psychometrics-sleep"`;
- substance use instruments such as `pm-auditc`, `pm-audit` and `pm-dast10` under `"psychometrics-longer"`;
- diary-type resources (e.g. mood and sleep diaries) under `"psychometrics-daily"`.

Each entry carries a stable `id` that matches the `LastLeafKey` mapping in `lastLeaf.index.ts` and the tags used by search and AI context builders. In other words, the psychometrics seed defines the "index spine" of the measurement-based care subtree, while the YAML/Markdown content provides the rich descriptions and example forms.

Psychotherapy scaffolds (`psychotherapies.ts`)

Psychotherapy content is more narrative and session-structured than raw scales. The `psychotherapies.ts` seed therefore starts from a higher-order source, `PSYCHOTHERAPIES` in `../content/psychotherapies`, and transforms each entry into a full `Card` with prompts and evidence:

```
export function buildPsychotherapyCards(): Card[] {
  return PSYCHOTHERAPIES.map<Card>((it) => {
    const baseTags: string[] = ['cbt','psychotherapy'];
    const t = it.title.toLowerCase();
    if (t.includes('insomnia') || t.includes('cbt-i'))
      baseTags.push('insomnia','sleep');
    if (t.includes('exposure')) baseTags.push('anxiety');
    if (t.includes('ptsd')) baseTags.push('ptsd');
    if (t.includes('ocd')) baseTags.push('anxiety');

    const prompts: PromptSpec[] = (it.prompts || []).map((text, idx) => ({
      id: `p${idx + 1}`,
      label: `Prompt ${idx + 1}`,
      text
    }));

    const examples = (it.examples || []).map((ex, idx) => ({
      id: `e${idx + 1}`,
      label: ex.label ?? `Example ${idx + 1}`,
      html: ex.html
    }));
```

```
    const evidence = (it.evidence || []).map((e) => ({
      title: e.title,
      authors: e.authors,
      year: e.year,
      journal: e.journal,
      doi: e.doi,
      link: e.link,
      level: e.level,
      note: e.note
    }));

    const summary =
      it.clinical_summary?.[0] ?? it.title;

    return {
      id: it.id,
      title: it.title,
      sectionId: 'psychotherapy',
      summary,
      tags: baseTags as any,
      examples,
      prompts,
      evidence
    } as Card;
  });
}
```

This seed produces cards such as structured CBT group programs, trauma-focused protocols and OCD exposure hierarchies. Each card is anchored to the `"psychotherapy"` section, tagged by modality and clinical focus (e.g. `"insomnia"`, `"ptsd"`, `"anxiety"`) and wired to prompts that can be sent directly to the AI assistant (e.g. "draft a session plan based on this scaffold for a 6-week CBT-i group").

Medication selection guidance (`medicationSelection.ts`)

The `medicationSelection.ts` seed builds a domain-specific library for antidepressant and related pharmacotherapy decisions. It imports `MEDICATION_SELECTION` from `../content/medicationSelection` and maps each item into a `Card` under the `"medications"` section:

```
export function buildMedicationSelectionCards(): Card[] {
  return MEDICATION_SELECTION.map<Card>((it) => {
    const baseTags: string[] = ['medications','ssri','snri'];
    const t = it.title.toLowerCase();
    if (t.includes('perinatal') || t.includes('pregnancy')
        || t.includes('lactation')) baseTags.push('perinatal');
    if (t.includes('geriatric') || t.includes('older')
        || t.includes('anticholinergic')) baseTags.push('geriatric');
    if (t.includes('augmentation') || t.includes('switch'))
      baseTags.push('augmentation');
```

```
    if (t.includes('cyp') || t.includes('qt'))
      baseTags.push('interactions','qt');

    const prompts: PromptSpec[] = (it.prompts || []).map((text, idx) => ({
      id: `p${idx + 1}`,
      label: `Prompt ${idx + 1}`,
      text
    }));

    const examples = (it.examples || []).map((ex, idx) => ({
      id: `e${idx + 1}`,
      label: ex.label ?? `Example ${idx + 1}`,
      html: ex.html
    }));

    const evidence = (it.evidence || []).map((e) => ({
      title: e.title,
      authors: e.authors,
      year: e.year,
      journal: e.journal,
      doi: e.doi,
      link: e.link,
      level: e.level,
      note: e.note
    }));

    const summary =
      it.clinical_summary?.length ? it.clinical_summary[0] : it.title;

    return {
      id: it.id,
      title: it.title,
      sectionId: 'medications',
      summary,
      tags: baseTags as any,
      examples,
      prompts,
      evidence
    } as Card;
  });
}
```

Clinically, this seed underpins cards for SSRI/SNRI selection, perinatal risk-benefit discussions, geriatric considerations (anticholinergic burden, falls, QTc) and augmentation strategies. Tags such as `"perinatal"`, `"geriatric"`, `"augmentation"` and `"interactions"` allow the AI panel and search UI to quickly surface the relevant subset for a given patient profile.

Population- and context-specific bundles

The remaining seed modules follow the same pattern but target specific populations or documentation needs:

- **camhs.ts** constructs cards under the "camhs" section, covering child and adolescent assessments, parent and teacher forms, and developmentally sensitive psychoeducation.
- **neuroMed.ts** builds "neuro-medical" cards for neurology–psychiatry liaison, cognitive screening, and medication interactions with comorbid neurological conditions.
- **caseLetters.ts** generates cards in the "case-forms-letters" section, including work letters, legal summaries and travel or fitness-to-drive certificates, each with structured prompts and evidence notes.
- **groupsPrograms.ts** targets the "groups-programs" section, providing group program overviews, attendance/monitoring scaffolds and example group psychoeducation sequences.

Internally, these modules share a common idiom:

$$\text{seedToCard} : \mathcal{S} \to \text{Card}, \tag{5.2}$$

$$\text{seedToCard}(s) = \big(\text{id}(s),\ \text{title}(s),\ \text{sectionId}(s), \tag{5.3}$$

$$\text{summary}(s),\ \text{tags}(s),\ \text{examples}(s), \tag{5.4}$$

$$\text{prompts}(s),\ \text{evidence}(s)\big). \tag{5.5}$$

where $\mathcal{S}$ is a seed definition space (CAMHS, neuropsychiatry, etc.) and $\text{sectionId}(s)$ is a deterministic mapping to one of the SectionId discriminants defined in ContentSchema.ts.

Table 18: *Overview of seed modules and their primary clinical domains.*

Seed module	Target SectionId(s)	Typical clinical content
psychometrics.ts	"psychometrics-*", "scales-measures"	Indexed list of scales (e.g. PHQ-9, GAD-7, PSQI, AUDIT), diaries and autoscore tools used in routine outcome monitoring.
psychotherapies.ts	"psychotherapy"	Session-structured CBT and other modalities, group protocols, example narratives, prompts and evidence panels.
medicationSelection.ts	"medications"	Antidepressant selection guides, perinatal and geriatric considerations, augmentation and switching strategies, and interaction summaries.
camhs.ts, neuroMed.ts	"camhs", "neuro-medical"	Child and adolescent pathways, neuropsychiatry liaison, cognitive screening and medical comorbidity anchors.
caseLetters.ts, groupsPrograms.ts	"case-forms-letters", "groups-programs"	Standardised letters, certificates and group program scaffolds with promptable templates and evidence references.

Legacy library integration (legacyLibrary.ts)

Not all content is yet migrated to the new YAML-based pipeline. The legacyLibrary.ts module provides a compatibility layer that merges older card collections with seed-based libraries:

```
import type { Card } from './lib/types';
import { mbcCards } from './legacy/mbc';
import { rapidTriageCards } from './legacy/rapidTriage';
import { medicationOrderCards } from './legacy/medicationOrders';
import { buildPsychotherapyCards } from './seeds/psychotherapies';
import { buildMedicationSelectionCards } from './seeds/medicationSelection';
import { buildPsychometricsCards } from './seeds/psychometrics';
```

```
export function buildLegacyLibrary(): Card[] {
  return [
    ...mbcCards,
    ...rapidTriageCards,
    ...medicationOrderCards,
    ...buildPsychotherapyCards(),
    ...buildMedicationSelectionCards(),
    ...buildPsychometricsCards()
  ];
}
```

Here, `mbcCards`, `rapidTriageCards` and `medicationOrderCards` are pre-existing card arrays authored before the introduction of the YAML schema. The seed functions bridge new domains into the same interface, allowing the rest of the system (UI, AI panel, export tools) to operate on a unified `Card[]` abstraction.

Content evolution, versioning and multi-locale support

The long-term strategy is to treat seeds as a high-level, typed interface on top of a versioned content store. Three design elements are important:

Versioning. At the YAML level, each document carries a `schema_version` field (Section 5.3). Seed modules are written so that their outputs are fully compatible with the `Card` and `Library` types regardless of schema upgrades. When a breaking change is required, a new `schema_version` is introduced in the YAML files, while seeds are gradually refactored to produce the richer structures (for example, adding structured scoring rules for scales or explicit risk grade fields).

Multi-locale content (`LocaleCode`). The `LocaleCode` discriminant in `ContentSchema.ts` is defined as

```
export type LocaleCode = "en" | "tr" | (string & {});
```

This ensures that both YAML-based cards and seed-based cards can be extended to additional locales without changing the type surface. Seed modules currently emit English titles and summaries by default; however, they are designed to work alongside `i18n` blocks in YAML documents so that future multi-locale builds can override titles, summaries and prompt labels while preserving stable `id` and `sectionId` values.

Future extensions. New domain seeds (e.g. perinatal psychiatry, pain management, addiction medicine) can be added by following the same pattern:

$$\texttt{DomainSeed.ts}: \quad \mathcal{D}_{\text{domain}} \xrightarrow{\text{seedToCard}} \mathcal{C}_{\text{domain}} \subseteq \texttt{Card[]}.$$

Because downstream consumers (`Library`, right panel bundles, AI context builders) only depend on the `Card` interface and the `SectionId` discriminant, these new modules plug into the existing infrastructure without further changes. Over time, legacy arrays in `./legacy/*` can be retired as their content is re-authored in YAML or seed-backed form, moving the entire psychiatry library towards a transparent, versioned and locale-aware knowledge base.

5.5 Psychiatry Modal UI and Store

The psychiatry toolkit is presented to the clinician as a full-screen modal shell implemented in `PsychiatryModal.tsx`. This shell is responsible for (i) loading and injecting the merged psychiatry library, (ii) wiring the left rail, center panel, and right panel to a shared Zustand store, and (iii) exposing a coherent keyboard- and screen-reader-friendly surface that sits on top of the generic SynapseCore center panel.

Modal lifecycle and content loading

On module load, `PsychiatryModal.tsx` eagerly initialises the content library:

- *Content library.* A type-safe `ContentLibrary` is loaded via `loadPsychContent()`, which executes the YAML/Markdown pipeline described in Sections 5.2 and 5.3.
- *Seeded and legacy cards.* Seed modules (`camhs.ts`, `groupsPrograms.ts`, `caseLetters.ts`, `neuroMed.ts`) and the legacy library builder are invoked to produce a unified `Card[]` collection. Content with identical identifiers in both pipelines is deduplicated by constructing a set of content identifiers and injecting only non-overlapping legacy entries.
- *Merged library.* The resulting merged array ($\mathcal{L} = \{c_1, \ldots, c_n\}$) is stored in a module-level variable `LIBRARY`. A small reordering step places preferred intake and mental status cards at the top, ensuring that the first focusable card is clinically meaningful.

When the React component is mounted, a guarded effect injects the library into the psychiatry store via `__setPsychLibrary`, but only once per runtime:

```
export default function PsychiatryModal({ open, onClose }: Props) {
  const injectedRef = useRef(false);

  useEffect(() => {
    if (!injectedRef.current && LIBRARY.length > 0) {
      __setPsychLibrary(LIBRARY);
      injectedRef.current = true;
    }
  }, []);

}
```

The modal accepts an optional `open` prop; if omitted, visibility is controlled entirely via the internal store. Closing the modal triggers a short fade-out ($\approx$ 300 ms) before either dispatching `store.close()` or calling the parent `onClose` callback. A global `synapse:ui:close` event listener allows other parts of SynapseCore (for example, the IDE header) to programmatically dismiss the psychiatry workspace.

Shared state model in `store.ts`

The central psychiatry store is implemented with Zustand in `store.ts`. The core shape is captured by `PsychStateV2`:

```
export type PsychStateV2 = {
  isOpen: boolean;
  section: PsychSection | "All";
```

```
  query: string;
  selectedCardId: string | null;
  favorites: string[];
  recentlyUsed: string[];
  riskFlag: boolean;
  settings: PsychSettings;
  uiPrefs: UiPrefs;
};
```

The store is persisted under a dedicated local-storage key (`"psychiatry.v2"`). On every mutation, a lightweight `saveV2` helper serialises a small subset of the state (`section`, `query`, `selectedCardId`, `favorites`, `recentlyUsed`, `riskFlag`, `settings`, `uiPrefs`), allowing clinicians to return to their previous card and filters.

From a formal perspective, the psychiatry state at time t can be seen as a vector

$$x_t = \big(s_t,\ q_t,\ T_t,\ i_t,\ F_t,\ R_t, \theta_t, v_t\big),$$

where s_t is the current section, q_t the search query, T_t the active tag set, i_t the selected card identifier, F_t the favourite list, R_t the recency list, θ_t the settings bundle, and v_t the UI preferences.

The filtering hook `usePsychFilter` implements a deterministic map from this state to a visible subset of cards:

$$\mathcal{F}(x_t) = \big\{c \in \mathcal{L} : \phi_{\text{section}}(c, s_t) \wedge \phi_{\text{text}}(c, q_t) \wedge \phi_{\text{tags}}(c, T_t) \wedge \phi_{\text{fav}}(c, F_t)\big\},$$

where the ϕ predicates encode section membership, search matches, tag intersections, and optional favourite-only mode. When "recommendation mode" is active, the hook additionally reorders $\mathcal{F}(x_t)$ by a score computed from the recency vector R_t and a simple content-based similarity signal derived from card metadata.

Selectors such as `selectSelectedCardId` and `selectSelectedCard` provide a narrow interface for the UI:

```
export const selectSelectedCardId = (s: PsychStore) => s.selectedCardId;
export const selectSelectedCard = (s: PsychStore) =>
  (s as any).getSelectedCard?.() ?? null;

export const useSelectedCardId = () => usePsychStore(selectSelectedCardId);
export const useSelectedCard   = () => usePsychStore(selectSelectedCard);
```

The modal uses `useSelectedCardId()` to determine whether a card is already selected; if not, it promotes the first filtered card (or the first entry in `LIBRARY`) to become the current card, ensuring that the right panel and editor always have a concrete target.

Debug instrumentation

`storeDebug.ts` contains an opt-in diagnostic harness that activates when the URL includes the `psychDebugStore` query parameter. It subscribes to store updates and counts the number of transitions within sliding time windows of length $\Delta t = 1$ s. Let u_t denote the number of updates in the interval $[t, t + \Delta t)$. If

$$u_t > \tau, \qquad \tau = 40,$$

a warning with summary statistics (update rate, sampling window, representative keys) is emitted to the browser console. The unsubcribe handle is exposed as `window.__psychStoreUnsub` for manual cleanup during debugging sessions.

Layout: header, navigation rail, center panel, right panel

The visible surface of the psychiatry modal is built as a grid with three vertical slices (left rail, center panel, right panel) beneath a shared header and welcome overlay:

- *Header and welcome flow.* The top row is handled by `TopHeader.tsx`, which renders the view-mode switcher (card, prompts, evidence), global search entry, CTA buttons (copy, send, insert, print) and shortcut handlers. A separate `WelcomeModal.tsx` offers an introductory description of the toolkit, emphasising that it is intended for clinicians and linking to the main Synapse IDE repository. The welcome modal is rendered via a portal, with a short close animation and appropriate ARIA attributes.
- *Left rail.* The left column hosts `RailContainer`, which uses `SECTION_TREE` and the psychiatry store to display grouped sections, search filters, favourites and tag chips. It consumes fields such as `section`, `query`, `activeTags`, `favorites` and exposes callbacks for toggling tags, navigating between sections and clearing filters. Internally, the rail integrates a fuzzy search index (via Fuse.js) to support tolerant matching of card titles.
- *Center panel.* The middle column embeds the generic `CenterPanelShell` as `CenterPanel`, configured with a title, subtitle and an `OutlineNav` slot. This reuse ensures that psychiatry content inherits the same outline navigation, scrolling semantics and persistence layer as the rest of SynapseCore.
- *Right panel.* The right column lazily mounts `RightPanelBoundary`, which wraps the four-block layout component (`RightPanelFourBlock.tsx`) and a registry defined in `rightPanelRegistry.ts`. For each selected card, the boundary resolves the appropriate content bundle (psychotherapy, medication selection, follow-up monitoring, scales, patient handouts, letters, etc.), constructs the four-block view (information, prompts, examples, evidence), and exposes export operations such as copy, download, print and open-in-new-tab.

The orchestration between these parts can be summarised schematically.

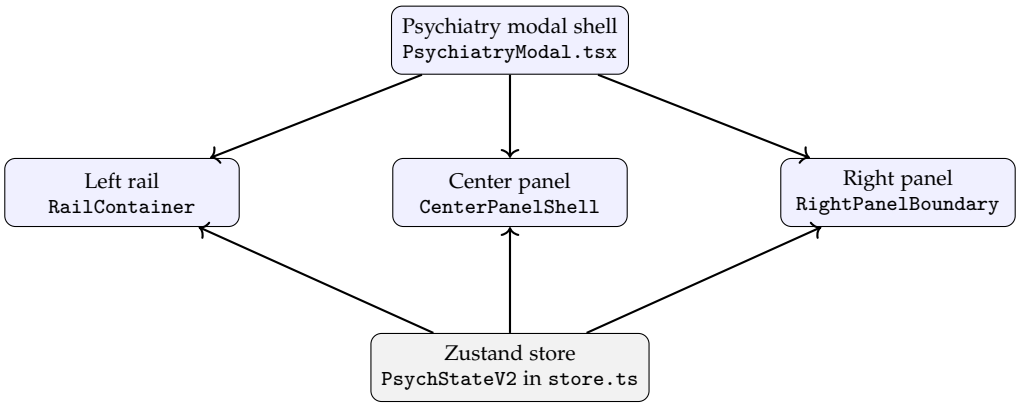


Figure 9: *High-level wiring between the psychiatry modal shell, shared store, and the three primary UI columns.*

Store slices and UI consumers

Table 19 summarises the main conceptual slices of the psychiatry store and the UI components that rely on them. The implementation uses a single Zustand store with a combined state

type, but selectors and hooks are arranged so that each slice can evolve independently.

Table 19: *Conceptual slices of the psychiatry store and primary UI consumers.*

Slice / selector	Key fields	Primary consumers
Navigation and filters	`section`, `query`, `activeTags`, `favOnly`, `recMode`	`RailContainer` (section tree, search), header search, `usePsychFilter` and badge counts in the left rail.
Selection and recency	`selectedCardId`, `recentlyUsed`, `getSelectedCard()`	Core modal (default selection logic), `RightPanelBoundary` and the editor region, which all need a concrete card context.
Favourites and risk flag	`favorites`, `toggleFavorite()`, `riskFlag`, `setRiskFlag()`	Favourite stars in the rail, "favourites-only" view, and future safety overlays which highlight risk-related content bundles.
Settings and UI preferences	`settings` (e.g. autosave, export behaviour), `uiPrefs` (density, right pane width)	`TopHeader` settings dialog, layout density toggles, and CSS-driven adjustments to the three-column shell.

This architecture keeps the psychiatry toolkit clinically coherent: the modal shell can be opened from anywhere in SynapseCore, yet its internals remain a single, well-defined store $\mathcal{S} = \{x_t\}$ with clear, typed interfaces to the left rail, center panel, and right panel content engines.

5.6 Evidence and Prompt Integration

The psychiatry AI layer couples curated prompts, explicit evidence snippets, JSON Schema constraints and export utilities into a single, measurement-aware workflow. For a fixed encounter, the effective AI request can be abstracted as

$$y = \Phi\big(c,\, p,\, \mathcal{E}_c,\, \sigma\big),$$

where $c \in \mathcal{L}$ is the selected content card (Section 5.4), p is a domain-specific prompt scaffold, $\mathcal{E}_c$ is the set of evidence items attached to that card, and σ is an optional JSON Schema describing the intended structure of the reply. The modules `components/ai/psychiatry/prompts.ts`, `components/ai/psychiatry/schemas.ts` and `components/ai/psychiatry/utils/exporters.ts` provide the corresponding building blocks.

Prompt scaffolds and clinical frames

The `prompts.ts` module declares a set of reusable textual scaffolds that encode widely used psychiatric documentation and assessment frames. Each scaffold is exported as a constant string and referenced from the psychiatry catalog (`packs.ts`; Section 5.4) via a `PsychiatryItem` of kind `'prompt'`.

The core example is the `MSE` scaffold, which follows the standard mental status examination headings: appearance, behaviour, speech, mood, affect, thought process, thought content, perception, cognition, insight, judgement and risk. Instead of attempting to fill these fields automatically, the scaffold instructs the AI to generate a structured template that the clinician can complete or adapt. In other words, the AI output is a *frame* for clinical thinking, not a substitute for the examination itself.

In addition, `prompts.ts` exposes:

- guideline index prompts such as `EVIDENCE_APA`, which enumerate major topics from professional guidelines (for example, depressive disorders, substance use disorders,

schizophrenia, PTSD, bipolar disorder and suicide risk assessment) and explicitly ask the AI to treat them as educational indices rather than prescriptive guidance;
- the constant `GAD7_JSON_SCHEMA`, which is technically a JSON Schema object but plays the same role in the catalog as textual scaffolds (see below).

The psychiatry catalog (`psychiatryCatalog` in `packs.ts`) groups these scaffolds into packs such as *Screeners*, *Safety & Risk*, *Psychoeducation* and *Scoring Code*. Each entry has a stable `id`, human-readable `label`, and tag set; this allows the AI sidebar in the Psychiatry Modal (Section 5.5) to filter prompts by clinical task (for example, `"phq9"`, `"risk"`, `"patient"`, `"instructions"`).

Table 20 summarises the main prompt families without reproducing the full catalog.

Table 20: *Conceptual categories of psychiatry prompt scaffolds.*

Prompt family	Representative scaffolds	Intended clinical use
Core documentation	MSE scaffold; SOAP-style outline; generic HPI and follow-up note frames	Structuring standard sections of psychiatric encounters (mental status examination, HPI, progress notes) in a reproducible way.
Risk, capacity, safety	Risk triage summaries; capacity assessment outlines; safety-plan prompts	Supporting narrative documentation of acute and longitudinal risk management, including protective factors and means safety planning.
Psychoeducation and handouts	Sleep hygiene / insomnia psychoeducation; depression and anxiety handout scaffolds; trauma-informed care summaries	Generating or refining patient-facing explanations and self-help materials, while keeping the final wording clinician-edited.
Guideline indices and evidence anchors	Educational indices for DSM–5–TR, NICE, WHO mhGAP, APA practice guidelines (including `EVIDENCE_APA`)	Anchoring AI suggestions to recognised frameworks and reminding users to consult official sources for up-to-date recommendations.

Operationally, the psychiatry packs use these scaffolds in conjunction with card-level evidence (for example, `Card.evidence` lists for specific scales or treatments). When a clinician selects a pack item, the system concatenates the scaffold with any relevant evidence snippets $\mathcal{E}_c$ and the patient-specific context (for example, PHQ–9 item responses) before sending the request to the selected AI route.

JSON Schemas and constrained numeric outputs

The `schemas.ts` and `prompts.ts` modules together provide JSON Schema descriptors for instruments that produce bounded integer responses. At present, there are two central schemas:

- `PHQ9_JSON_SCHEMA` in `schemas.ts`, covering nine depression items ($q_1, \ldots, q_9$);
- `GAD7_JSON_SCHEMA` in `prompts.ts`, covering seven generalised anxiety items ($q_1, \ldots, q_7$).

Both schemas share the same structural pattern:

- the root type is `"object"` with `"additionalProperties": false`;
- each property `"qk"` is constrained to integers between 0 and 3 inclusive;
- all item keys are required.

Formally, for an instrument with n items, the admissible answer set is

$$\Omega_n = \left\{ \mathbf{a} \in \mathbb{Z}^n \;:\; 0 \le a_i \le 3 \text{ for all } i = 1, \ldots, n \right\}.$$

The JSON Schema σ_n enforces membership in Ω_n at the level of the AI response; any deviation (wrong length, out-of-range value, non-integer) leads to validation failure and a visible error rather than a silently incorrect score.

The scoring helpers in `score.ts` then map validated response vectors to a total score and severity band. For the PHQ–9, the total score is

$$S_{\text{PHQ9}}(\mathbf{a}) = \sum_{i=1}^{9} a_i,$$

with $a_i \in \{0, 1, 2, 3\}$. The severity mapping g_{PHQ9} takes total scores $s \in \{0, \ldots, 27\}$ to the band labels "minimal", "mild", "moderate", "moderately severe" and "severe" and follows the standard cut-points:

$$g_{\text{PHQ9}}(s) = \begin{cases} \text{minimal}, & 0 \le s \le 4, \\ \text{mild}, & 5 \le s \le 9, \\ \text{moderate}, & 10 \le s \le 14, \\ \text{moderately severe}, & 15 \le s \le 19, \\ \text{severe}, & 20 \le s \le 27. \end{cases}$$

An analogous mapping g_{GAD7} is implemented for the GAD–7 with range 0–21. The TypeScript and Python code in `score.ts` mirrors this piecewise definition and performs the same input checks, making it suitable for direct inclusion in Jupyter notebooks or analytic pipelines.

This architecture has two important consequences:

1. It decouples narrative generation from numeric scoring. The same PHQ–9 responses can be used both for AI-assisted narrative risk summaries and for strictly local computation of the total score and severity band.
2. It makes any future psychometric instrument amenable to the same pattern. As long as the response space is finite and can be expressed as a conjunction of constraints on each item, a new JSON Schema σ can be added with minimal changes to the rest of the system.

Exporting AI-generated artefacts

The module `components/ai/psychiatry/utils/exporters.ts` defines a compact set of helper functions to move AI-generated artefacts (for example, risk narratives, letters, HTML forms and code snippets) across the boundary between the browser and the outside world, while keeping the core SynapseCore runtime agnostic about specific clinical formats.

Two low-level helpers convert between languages, file extensions and MIME types:

- `extFromLang(lang?)` maps language labels (e.g. `typescript`, `javascript`, `python`, `html`) to file extensions (`ts`, `js`, `py`, `html`), defaulting to `txt` for unknown or plain-text payloads.
- `mimeFromExt(ext)` maps file extensions to MIME types. For example:
  `.html` $\rightarrow$ `text/html;charset=utf-8`,
  other extensions $\rightarrow$ `text/plain;charset=utf-8`.

The omitted middle section (indicated by `...` in the source) is responsible for constructing `Blobs`, opening new tabs, triggering downloads and optionally copying text to the clipboard. Conceptually, it implements functions of the form

$$\Psi : (\text{text},\ \text{filename},\ \text{mime}) \longrightarrow \{\text{tab},\ \text{download},\ \text{clipboard}\},$$

where the codomain is interpreted as a side-effect (for example, opening a new tab showing a printable HTML letter).

The final helper, `sharePayload`, adds an optional remote-sharing layer. If the build-time environment variable `VITE_PSYCH_SHARE_ENDPOINT` is defined, the function performs a `fetch()` call to that endpoint with a JSON body containing:

- the suggested file name (`filename`);
- the MIME type (`mime`);
- the raw text (`text`);
- a time-to-live in days (`ttlDays`, currently 7);
- a visibility flag (currently `'unlisted'`).

If the server returns a JSON object with a field `url`, it is interpreted as a short-lived share link and returned to the caller. If the environment variable is absent or the request fails, the function falls back to a URL-encoded `data:` URL:

$$u_{\text{fallback}} = \texttt{`data:}\mathit{mime}\texttt{,encodeURIComponent(text)`},$$

which can still be opened in a new tab or bookmarked locally. This design ensures that export functionality never fully depends on a remote service: the remote share is a progressive enhancement over a purely local baseline.

End-to-end interaction: from card and evidence to artefact

Combining the pieces above with the psychiatry modal store (Section 5.5), the end-to-end AI interaction can be described as a sequence:

1. The clinician selects a card c in the Psychiatry Modal (for example, "PHQ–9 follow-up", "Generalised anxiety review" or "Safety plan").
2. The UI computes the associated prompt scaffold p and collects relevant evidence items $\mathcal{E}_c$ from the card metadata and guideline indices.
3. If the interaction involves an instrument (for example, PHQ–9), the appropriate JSON Schema σ is attached to the request and the AI is asked to return a strictly structured payload.
4. The AI route processes the request and returns a response y. For structured replies, y is validated against σ; for narrative replies, it is treated as rich text.
5. Structured numerical data are passed to scoring helpers (for example, `scorePHQ9`) and, if appropriate, to visual summaries elsewhere in SynapseCore. Narrative data are presented in the right panel and can be edited by the clinician.
6. When the clinician is satisfied, the export helpers are used to copy, print, download or (where configured) share the final artefact.

This pipeline keeps three invariants explicit: (i) prompts and evidence are always visible to the clinician, (ii) numeric outputs for scales are guarded by JSON Schemas and transparent scoring functions, and (iii) exported artefacts are under clinician control rather than being pushed directly into external records. From a design perspective, the psychiatry AI layer therefore behaves as an *evidence-anchored assistant* embedded in the clinical workflow, rather than an opaque black-box generator.

6 Measurement-Based Care Engine

Measurement-based care (MBC) denotes the systematic and repeated collection of validated symptom, functioning, and side-effect measures to inform clinical decision-making in real time (Fortney et al., 2017a; Guo et al., 2015). In SynapseCore, the MBC engine is designed as

a first-class subsystem: it must accept item-level responses for multiple instruments, compute scores and severity bands in a fully deterministic way, attach clinically meaningful flags, and expose all of this as well-typed objects that can be safely consumed by flows, AI components, and export pipelines.

The implementation aims to reconcile three constraints:

C1 *Psychometric fidelity.* Scoring rules follow instrument manuals as closely as possible (item domains, cut-offs, cluster criteria), so that SynapseCore reproduces the behaviour expected from paper-based scoring (Bush et al., 1998; Goodman et al., 1989; Kroenke et al., 2001a; Spitzer et al., 2006; Weathers et al., 2013).

C2 *Computational robustness.* All numerical operations are pure, bounded, and idempotent. In particular, the same input vector always yields the same score and flags, regardless of the calling context.

C3 *Reusability and extensibility.* A single schema must support both simple screeners (e.g. PHQ-9) and more structured instruments (e.g. PCL-5 cluster logic), and allow new scales to be plugged in without modifying downstream consumers.

These requirements are crystallised in the small but expressive schema defined in `src/features/psychiatry/mbc/calculators.ts`, structured around the core types `ScoreBand` and `ScoreResult` and the helper functions `clamp`, `sum`, and `coerce`.

6.1 ScoreResult and band schema

Type-level schema

At the TypeScript level the scoring schema is given by:

$$\text{ScoreBand} := \{\, (\ell, m, M) \mid \ell \in \mathcal{L},\ m, M \in \mathbb{N},\ m \le M \,\}, \tag{6.1}$$

$$\text{ScoreResult} := (S, \sigma, \mathcal{B}, F, \beta), \tag{6.2}$$

where:

- $\mathcal{L}$ is the set of human-readable band labels (e.g. “None”, “Mild”, “Moderately severe”),
- $S \in \mathbb{N}$ is the total scale score,
- $\sigma \in \mathcal{L}$ is the selected severity band label,
- $\mathcal{B} = (b_1, \ldots, b_K)$ is an ordered list of bands with $b_k = (\ell_k, m_k, M_k) \in \text{ScoreBand}$,
- $F \subseteq \mathcal{F}$ is a finite set of string-valued flags for clinically salient patterns (e.g. “Item 9 > 0 — discuss safety today”),
- β is an optional breakdown object storing structured sub-scores or cluster indicators.

In the concrete code this reads:

```
export type ScoreBand = { label:  string; min:  number; max:  number
};
export type ScoreResult = {
   total:  number; severity:  string;
   bands:  ScoreBand[]; flags:  string[];
   breakdown?:  Record<string, unknown>;
};
```

This is deliberately minimal: all higher-level constructs in SynapseCore operate only on these fields, which ensures that new instruments can be added by shipping a single new scoring function that returns a `ScoreResult`.

Mathematical representation of item vectors

Each instrument is modelled as a discrete response vector

$$x = (x_1, \ldots, x_n) \in \{0, \ldots, k\}^n, \tag{6.3}$$

where n is the number of items and k is the highest admissible Likert category for that scale (e.g. $k = 3$ for PHQ-9 and GAD-7, $k = 4$ for PCL-5 and Y–BOCS). In practice, user interfaces and upstream systems may provide an array $\tilde{x}$ that is shorter, longer, or contains out-of-range values. Before any psychometric logic is applied, SynapseCore normalises this array via the helper `coerce(items, len, lo, hi)`, which can be expressed as:

$$\text{coerce}(\tilde{x}; n, \ell, h)_i = \begin{cases} \min\{h, \max\{\ell, \tilde{x}_i\}\}, & 1 \le i \le \min(n, |\tilde{x}|), \\ 0, & \text{otherwise.} \end{cases} \tag{6.4}$$

Here ℓ and h are the minimum and maximum admissible item values, and $|\tilde{x}|$ is the length of the raw array. Algebraically, coerce has three useful properties:

P1 *Boundedness:* $\ell \le \text{coerce}(\tilde{x})_i \le h$ for all i.
P2 *Idempotence:* $\text{coerce}(\text{coerce}(\tilde{x})) = \text{coerce}(\tilde{x})$.
P3 *Monotonicity:* if $\tilde{x}_i \le \tilde{y}_i$ for all i, then $\text{coerce}(\tilde{x})_i \le \text{coerce}(\tilde{y})_i$.

These properties guarantee that the same clinical state, represented by the same raw responses, will always lead to the same normalised vector, and therefore to the same score and flags, regardless of how many times or in which context the scoring function is called.

Score function and band mapping

Given a normalised response vector x, the core score function is defined as the discrete sum

$$S(x) \;=\; \sum_{i=1}^{n} x_i, \tag{6.5}$$

implemented in code via the helper `sum`. This immediately yields the bound

$$0 \;\le\; S(x) \;\le\; nk, \tag{6.6}$$

with the lower bound corresponding to no endorsed symptoms and the upper bound to maximum severity on all items.

Severity bands are specified as an ordered family of integer intervals $\{[m_j, M_j]\}_{j=1}^{K}$ with labels $\{\ell_j\}_{j=1}^{K}$, stored as an array of `ScoreBand` objects:

$$b_j = (\ell_j, m_j, M_j) \in \text{ScoreBand}.$$

Formally, the band mapping is a function

$$B : \mathbb{N} \to \mathcal{L} \cup \{\text{Unknown}\}, \qquad B(S) = \begin{cases} \ell_j, & \text{if } m_j \le S \le M_j \text{ for some } j, \\ \text{Unknown}, & \text{otherwise.} \end{cases} \tag{6.7}$$

Under well-specified bands, the union of intervals $\bigcup_{j=1}^{K} [m_j, M_j]$ covers the entire feasible score range $[0, nk]$, so that "Unknown" never occurs in practice. The `severity` field in `ScoreResult` is exactly $\sigma = B(S(x))$. Changing cut-offs or labels for an instrument thus reduces to editing the band array for that particular scorer; no consumer of the scoring engine needs to be modified.

Flags and breakdowns

Many instruments require additional logic beyond the scalar score and severity band. The MBC schema captures this via the `flags` array and the optional `breakdown` object:

- For the PHQ-9, any non-zero response on item 9 produces a safety flag, e.g. "Item 9 > 0 — discuss safety today", and high total scores may trigger "Severe depression — consider urgent review" (Kroenke et al., 2001a).
- For the PCL-5, SynapseCore computes cluster-level indicators $B, C, D, E \in \{\text{true}, \text{false}\}$ based on count thresholds within each symptom domain; these are stored in the `breakdown = {B,C,D,E}` field and a composite flag marks when cluster criteria are met (Weathers et al., 2013).
- For the AUDIT-C, item-specific thresholds (e.g. high values on the binge-drinking item) trigger dedicated flags, independent of the total score (Bush et al., 1998).

The design principle is that all such predicates are deterministic functions of the normalised item vector x. No probabilistic or model-based logic is applied at this stage: given x, the set of flags F and the breakdown β are uniquely determined. This keeps the MBC engine purely rule-based and auditable; any machine-learning components consuming these results must treat them as inputs, not as mutable state.

Implementation invariants

The most important invariants enforced by the schema are summarised in Table 21. These invariants are implicitly tested by unit tests over the internal helpers (`clamp`, `sum`, `coerce`) and by integration tests that compare SynapseCore scores against worked examples in the original scale manuals.

Table 21: *Key invariants of the MBC scoring schema. The table uses `tabularx` to remain within the page width.*

Component	Invariant
Total score S	For each instrument with n items and maximum per-item value k, the total score satisfies $0 \le S \le nk$. Scores are integer valued and computed solely from the normalised vector x.
Bands $\mathcal{B}$	Band intervals $[m_j, M_j]$ are integer-valued, closed, and either disjoint or only touching at boundaries. For all feasible scores $S \in [0, nk]$ there is at most one band with $m_j \le S \le M_j$.
Severity σ	The severity label is given by $\sigma = B(S)$ as in Equation (6.7). Changing clinical cut-offs is implemented exclusively by editing $\mathcal{B}$.
Flags F	Flags are generated by fixed predicates on x (e.g. item thresholds, cluster counts). Identical inputs always yield identical flag sets; no hidden state or randomness is involved.
Breakdown β	The optional breakdown object encodes additional structure (cluster booleans, domain sub-scores) derived deterministically from x. Its presence does not affect S or σ, but enables richer visualisations and AI prompts.

Pipeline overview

Figure 10 shows the logical pipeline from raw responses to the final `ScoreResult`. The pipeline is intentionally simple: all complexity (instrument-specific rules, cluster logic, safety thresholds) is contained within pure functions that have no side effects.

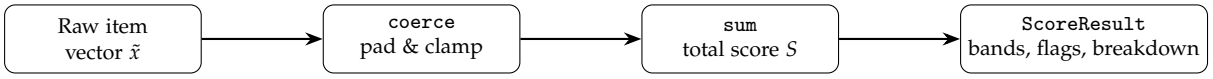


Figure 10: *Compact pipeline from raw responses $\tilde{x}$ to the `ScoreResult` object used by flows, the AI assistant, and export modules. The figure is horizontally compact to respect the article layout.*

6.2 Implemented Scales

Building on the generic `ScoreResult` schema introduced above, the current MBC engine instantiates a small but clinically dense set of instruments. Each instrument is implemented as a pure function that takes a vector of item responses and returns a `ScoreResult` with `total`, `severity`, ordered `bands`, and optional machine-readable `flags` and `breakdown` fields. This keeps all clinical logic (cut-points, caseness heuristics, safety flags) in a single, testable module (`mbc/calculators.ts`) instead of being scattered across UI components or prompt templates.

Table 22: *Implemented scales in the MBC engine: structural overview.*

Instrument	Items	Item range	Total range	Severity levels (labels)
PHQ–9	9	0–3	0–27	5-level: None, Mild, Moderate, Moderately severe, Severe.
GAD–7	7	0–3	0–21	4-level: None, Mild, Moderate, Severe.
PCL–5	20	0–4	0–80	2-level: Subthreshold vs. Probable PTSD (screen).
Y–BOCS	10	0–4	0–40	5-level: Subclinical, Mild, Moderate, Severe, Extreme.
AUDIT–C	3	0–4	0–12	2-level: Low vs. Screen positive (sex-specific cut-points).

Table 23: *Implemented scales in the MBC engine: flags, breakdowns, and primary usage.*

Instrument	Flags	Breakdown / primary usage
PHQ–9	Safety flag when item $9 > 0$: "Item $9 > 0$ — discuss safety today".	No cluster breakdown; primary usage is longitudinal depression monitoring and safety-triggering prompts.
GAD–7	None in the current release.	No breakdown; used as a parsimonious anxiety index and for stepped-care gating (e.g. when to escalate beyond psychoeducation).
PCL–5	"Total ≥ 33 — positive screen" and a cluster flag when DSM–5 B/C/D/E criteria are met.	`breakdown` encodes booleans for B, C, D, E clusters; used for PTSD screening, documenting caseness logic, and AI prompts that need explicit trauma topology.
Y–BOCS	No flags.	No breakdown; bands provide a scalar measure of OCD severity for tracking and summarisation (e.g. response vs. non-response).
AUDIT–C	Flag for heavy episodic use: "Q3 ≥ 4 — heavy episodic use flag".	No breakdown; sex-specific cut-points and episodic flag distinguish chronic high use from binge-pattern risk.

Internally, all calculators share the same defensive preprocessing routine `coerce(items, len, lo, hi)`, which truncates the array to a fixed length, fills missing entries with zero, and clamps non-finite values into the allowed range. This makes each scoring function a total function (items $\to$ `ScoreResult`) and avoids subtle UI-dependent corner cases (for example, partially answered forms, free-text inputs, or values pasted from spreadsheets).

Formally, for an instrument m with n_m items and per-item bounds $[l_m, u_m]$, the scoring map implemented in `mbc/calculators.ts` can be written as

$$f_m : [l_m, u_m]^{n_m} \longrightarrow \texttt{ScoreResult} = \{(\texttt{total}, \texttt{severity}, \texttt{bands}, \texttt{flags}, \texttt{breakdown})\},$$

where `severity` is determined by a banding function $\beta_m : \mathbb{Z} \to \{\text{band labels}\}$ and `flags` is a finite set of strings encoding clinically salient conditions.

PHQ–9 (Patient Health Questionnaire–9). PHQ–9 is implemented as a nine-dimensional Likert vector $\mathbf{x} = (x_1, \ldots, x_9)$ with item values in $\{0, 1, 2, 3\}$, corresponding to the standard frequency anchors from "not at all" to "nearly every day". After coercion,

$$\mathbf{x} = \texttt{coerce(items, 9, 0, 3)},$$

the total score is computed as

$$S_{\text{PHQ9}}(\mathbf{x}) = \sum_{i=1}^{9} x_i, \quad 0 \le S_{\text{PHQ9}} \le 27.$$

The severity banding function β_{PHQ9} is represented as an array of `ScoreBand` objects and corresponds to the canonical cut-offs:

$$\beta_{\text{PHQ9}}(s) = \begin{cases} \text{None}, & 0 \le s \le 4, \\ \text{Mild}, & 5 \le s \le 9, \\ \text{Moderate}, & 10 \le s \le 14, \\ \text{Moderately severe}, & 15 \le s \le 19, \\ \text{Severe}, & 20 \le s \le 27. \end{cases}$$

The implementation scans `bands` to find the unique element with $\texttt{min} \le S_{\text{PHQ9}} \le \texttt{max}$, and falls back to `'Unknown'` if no band matches (a defensive default for future extension).

Item 9, which captures thoughts of death or self-harm, is treated as a high-salience safety signal. Let x_9 denote the coerced value of item 9. The safety flag is generated according to

$$\mathbf{1}_{\text{risk}} = \begin{cases} 1, & x_9 > 0, \\ 0, & x_9 = 0, \end{cases}$$

and the corresponding string `'Item 9 > 0 -- discuss safety today'` is appended to the `flags` array when $\mathbf{1}_{\text{risk}} = 1$. This flag is consumed in several downstream locations: it is surfaced in autoscore HTML, carried into clinical note prompts, and available to external export pipelines without each consumer having to re-implement the safety heuristic.

GAD–7 (Generalized Anxiety Disorder–7). GAD–7 follows an almost identical pattern but with seven items. After coercion to $\mathbf{x} \in \{0, 1, 2, 3\}^7$, the total score is

$$S_{\text{GAD7}}(\mathbf{x}) = \sum_{i=1}^{7} x_i, \quad 0 \le S_{\text{GAD7}} \le 21.$$

The severity function β_{GAD7} implements the standard four-level categorisation:

$$\beta_{\text{GAD7}}(s) = \begin{cases} \text{None}, & 0 \le s \le 4, \\ \text{Mild}, & 5 \le s \le 9, \\ \text{Moderate}, & 10 \le s \le 14, \\ \text{Severe}, & 15 \le s \le 21. \end{cases}$$

In the present version, `gad7Score` does not emit any additional `flags`. This is an explicit design choice: for anxiety, the primary signals required by the rest of the system are (i) whether the patient is in the "moderate or above" range, and (ii) the direction of change over time. Both are captured by `total` and `severity`, and more fine-grained heuristics (for example, somatic vs. cognitive focus) can later be added without altering the overall interface.

PCL–5 (PTSD Checklist for DSM–5). PCL–5 is implemented as a twenty-item vector $\mathbf{x} \in \{0,1,2,3,4\}^{20}$ with the usual anchors from "not at all" (0) to "extremely" (4). After coercion,

$$S_{\text{PCL5}}(\mathbf{x}) = \sum_{i=1}^{20} x_i, \quad 0 \le S_{\text{PCL5}} \le 80.$$

The engine adopts a screening-oriented banding: subthreshold (0–32) vs. probable PTSD (33–80). While this is expressed in `bands`, the implementation uses a direct threshold check for `severity`:

$$\beta_{\text{PCL5}}(s) = \begin{cases} \text{Probable PTSD (screen)}, & s \ge 33, \\ \text{Subthreshold}, & s < 33. \end{cases}$$

Beyond the total, PCL–5 contributes a richer, cluster-based breakdown that mirrors DSM–5 criteria. A helper function

$$\text{atLeast}(A, n) = \big|\{a \in A : a \ge 2\}\big| \ge n$$

counts the number of items in a subset A that reach at least "2" (*moderately*) and compares it to the required number n. Using this, the calculators module defines

$$\begin{aligned} B &= \text{atLeast}(x_{1..5}, 1), \\ C &= \text{atLeast}(x_{6..7}, 1), \\ D &= \text{atLeast}(x_{8..14}, 2), \\ E &= \text{atLeast}(x_{15..20}, 2), \end{aligned}$$

and the overall caseness indicator

$$\texttt{clusterOK} = B \wedge C \wedge D \wedge E.$$

The `breakdown` field of `ScoreResult` stores the booleans $\{B, C, D, E\}$, and `flags` is enriched whenever $S_{\text{PCL5}} \ge 33$ and/or `clusterOK` is true. This structure allows the same scoring object to support: (a) simple PTSD screening via total score, (b) more nuanced documentation of DSM–5 criteria for clinical notes, and (c) AI prompts that explicitly reference which clusters are met without having to parse free-text.

Y–BOCS (Yale–Brown Obsessive–Compulsive Scale). Y–BOCS is represented as a ten-item vector with values in $\{0,1,2,3,4\}$. The coercion step enforces the correct length and bounds, and the total is

$$S_{\text{YBOCS}}(\mathbf{x}) = \sum_{i=1}^{10} x_i, \quad 0 \le S_{\text{YBOCS}} \le 40.$$

Severity bands are encoded as

$$\beta_{\text{YBOCS}}(s) = \begin{cases} \text{Subclinical}, & 0 \le s \le 7, \\ \text{Mild}, & 8 \le s \le 15, \\ \text{Moderate}, & 16 \le s \le 23, \\ \text{Severe}, & 24 \le s \le 31, \\ \text{Extreme}, & 32 \le s \le 40. \end{cases}$$

No additional `flags` or `breakdown` fields are attached in this release; the main goal is to provide a scalar OCD severity index that is easy to track longitudinally and to embed in narrative outputs (e.g. "Y–BOCS 28/40, severe range"). Because the band structure is explicit and stored alongside the total, future response/remission rules (for example, $\ge 35\%$ reduction plus endpoint below a threshold) can be implemented cleanly in a separate, longitudinal module without changing the scoring function itself.

AUDIT–C (Alcohol Use Disorders Identification Test, consumption subset). AUDIT–C is encoded as a three-item vector with responses in $\{0, 1, 2, 3, 4\}$. After coercion,

$$S_{\text{AUDITC}}(\mathbf{x}) = \sum_{i=1}^{3} x_i, \quad 0 \le S_{\text{AUDITC}} \le 12.$$

The banding is sex-specific. Let $g \in \{\text{F}, \text{M}, \text{Other}\}$ denote the sex parameter. The engine sets

$$c(g) = \begin{cases} 3, & g = \text{F}, \\ 4, & g \in \{\text{M}, \text{Other}\}, \end{cases}$$

and defines

$$\beta_{\text{AUDITC}}(s; g) = \begin{cases} \text{Screen positive}, & s \ge c(g), \\ \text{Low}, & s < c(g). \end{cases}$$

The `bands` array stores these ranges explicitly so they can be rendered in autoscore summaries. Additionally, item 3 (heavy episodic intake) is inspected, and when $x_3 \ge 4$ the calculators module appends the flag `'Q3 ≥ 4 -- heavy episodic use flag'`. This allows clinical notes and AI prompts to distinguish between persistent high weekly consumption and episodic binge patterns without duplicating logic.

Relationship to legacy MBC logic. Before the dedicated `mbc` module was introduced, scale handling in SynapseCore lived inside the UI-oriented `legacy/mbc.ts` file. There, card definitions mixed three concerns:

1. HTML forms with inputs like "PHQ–9 (Baseline)" and "PHQ–9 (Current)" for users to type in totals.
2. Embedded textual guidelines for response and remission (for example, "response $\ge$ 50% reduction from baseline; remission PHQ–9 ≤ 4 sustained over two visits").
3. Prompt templates that interpolated free-text fields such as `{{phq9}}` and `{{gad7}}` inside progress-note snippets.

This legacy design made the cards visually rich but also tightly coupled to manual scoring and prone to drift: a change in cut-points or a new flag needed to be reflected in multiple places (form labels, tooltip text, prompts), and there was no single authoritative arithmetic implementation.

The new `mbc/calculators.ts` module deliberately separates the concerns:

- All numeric rules, severity thresholds, and flags for PHQ–9, GAD–7, PCL–5, Y–BOCS, and AUDIT–C live in the calculators file and are exposed via a small, uniform API (`phq9Score`, `gad7Score`, `pcl5Score`, `ybocsScore`, `auditCScore`).
- Legacy cards still define forms and prompt templates, but they now consume structured results (via `ScoreResult` or `renderAutoscoreHTML`) instead of free-text totals.
- Response and remission heuristics described in the legacy HTML (for example, percentage reduction in PHQ–9) can be re-implemented in a future longitudinal analytics module that takes serial `ScoreResult` objects as input, keeping the core scoring strictly cross-sectional and easily testable.

Conceptually, the evolution is from "*templates that happen to mention scales*" to "*a central, typed scoring engine that templates can safely call*". This architecture allows the same implemented scales to serve three layers of the system simultaneously: MBC forms in the clinician UI, autoscore HTML used for immediate feedback and exports, and AI prompts that need reliable, machine-readable descriptions of symptom severity and risk.

6.3 Autoscore and HTML Reports

The measurement based care (MBC) calculators module does not only compute numeric scores. At the end of `src/features/psychiatry/mbc/calculators.ts` it exposes a compact HTML reporting utility, `renderAutoscoreHTML`, which converts a `ScoreResult` into a print ready mini report. From a computer science perspective this function is a pure, referentially transparent map

$$\text{renderAutoscoreHTML} : (\texttt{MeasureId}, \mathbf{x}, \texttt{RenderOpts}) \to \text{HTML}_{\text{string}},$$

where `MeasureId` identifies the instrument (`'phq9'`, `'gad7'`, `'pcl5'`, `'ybocs'`, `'auditc'`), $\mathbf{x}$ is the raw response vector, and `RenderOpts` carries small presentation hints such as a custom title, patient label, date string, and sex for banding (AUDIT–C). There is no hidden state; identical inputs always produce identical HTML.

Internally, the function delegates all numerical work to the scale calculators described in Section 6.2. For an instrument m with n_m items and item bounds $[l_m, u_m]$ the pipeline inside `renderAutoscoreHTML` is

$$\begin{aligned} \mathbf{x}_{\text{coerced}} &= \text{coerce}(\texttt{answers}, n_m, l_m, u_m), \\ r_m &= f_m(\mathbf{x}_{\text{coerced}}) = \{\texttt{total}, \texttt{severity}, \texttt{bands}, \texttt{flags}, \texttt{breakdown}\}, \\ h_m &= \text{renderTemplate}(m, r_m, \texttt{RenderOpts}), \end{aligned}$$

where f_m is one of `phq9Score`, `gad7Score`, `pcl5Score`, `ybocsScore`, or `auditCScore`. The final value h_m is a plain HTML string composed from a fixed template plus the values of r_m and the render options.

From a clinical viewpoint, the design goal is to give the clinician a consistent, low friction view of each scale: raw item scores, total score, severity band, and any risk flags are presented in a compact layout that can be read quickly during a session or attached to a letter.

Block level layout of the autoscore report. Instead of a free form HTML fragment, the output of `renderAutoscoreHTML` is organised into a small number of clearly separated blocks. This makes the report predictable for clinicians and easier to parse for downstream tools.

1. **Header block.** A `<section>` with a `<header><h2>` element that carries a short title, for example "PHQ-9 autoscore". If a custom title is supplied via `RenderOpts.title`, that value is used; otherwise the title is derived automatically from `MeasureId`.
2. **Patient identification line.** A paragraph beginning with `<strong>Patient</strong>` followed by a compact line that can be read when printed, such as "M. S., 35 y, f, 2025-11-22". The content is built from the free text `patient` label and `date` string in `RenderOpts`.
3. **Items table.** A short `<table>` introduced by an `<h3>Items</h3>` heading. Each row lists the item index and the final scored value after coercion. When the report is printed in black and white, this table provides a quick audit trail of how the total score was obtained.
4. **Computed summary.** A `<h3>Computed</h3>` heading followed by an unordered list. The first bullet displays the total score and the severity label. If `res.flags` is not empty, a second bullet is added of the form "Flags: f_1; f_2; ...; f_L".
5. **Severity anchors.** A short paragraph with an emphasised label "Severity anchors" followed by the exact band cut points. This allows a printed report to be interpreted without any external legend or manual.
6. **Footer note.** A final paragraph with a standardised caveat that scale scores are decision support and must be interpreted in the context of a full clinical assessment. The text is static and identical across instruments.

Anchors and flags as derived quantities. The anchors string is generated directly from the band definitions. Let $\mathcal{B}_m = \{(b_i^{\text{name}}, b_i^{\min}, b_i^{\max})\}_{i=1}^{K}$ denote the band metadata for instrument m. The anchors function

$$A_m : \mathcal{B}_m \to \text{string}$$

is implemented as

$$A_m(\mathcal{B}_m) = \text{join}\Big(\big\{b_i^{\text{name}} : b_i^{\min}\text{–}b_i^{\max}\big\}_{i=1}^{K}\Big),$$

where `join` uses a fixed separator such as " | ". This guarantees that the human readable cut points in the report are always consistent with the banding logic used by the calculators. If the bands are updated, both the numeric thresholding and the textual anchors change in lockstep.

Flags are rendered only when present. If `res.flags` $= [f_1, \ldots, f_L]$ is non empty, an additional bullet in the "Computed" section is appended:

$$\text{Flags: } f_1;\ f_2;\ \ldots;\ f_L.$$

Risk signals such as "Item 9 > 0 — discuss safety today" (PHQ-9) or "Q3 >= 4 — heavy episodic use flag" (AUDIT-C) therefore appear in a predictable position across all reports. From a psychiatry perspective this makes it easier to scan quickly for safety issues; from a systems perspective it means downstream export or AI layers do not have to parse free text scattered in different locations.

String level escaping inside `calculators.ts`. The HTML generator never trusts external strings. Instead it uses a small escaping function

$$\text{escapeHtml} : \text{string} \to \text{string},$$

which performs the substitutions

$$\text{escapeHtml}(s) = s \text{ with } \{\&, <, >, ''\} \text{ replaced by } \{\texttt{\&}, \texttt{\<}, \texttt{\>}, \texttt{\"}\}.$$

All strings that might originate outside the calculators module are passed through escapeHtml: the optional report title, the patient label, the date string, the severity label, the anchors string, and each flag. Only numeric values (item scores and totals) are inserted directly, because they are generated inside the calculators and constrained to small integer ranges. This separation keeps the mathematics of scoring (f_m) and the mechanics of HTML escaping clearly distinct, but tightly coupled by type: every path from user input to HTML crosses a dedicated escaping step.

Additional sanitisation in the psychiatry right panel. In practice the HTML generated by `renderAutoscoreHTML` is rarely inserted into the DOM verbatim. Instead, the psychiatry right panel applies a second layer of sanitisation in `src/features/psychiatry/rightPanelUtils.ts`. The function `sanitizeHtml` acts on a fragment h as follows:

1. Parse h into a detached document tree.
2. Remove elements that are not in a small allow list of presentational tags. Elements such as `<script>`, `<iframe>`, `<object>`, `<embed>`, `<link>` and `<meta>` are dropped completely.
3. Walk the remaining nodes and remove any attribute whose name begins with `on`, thereby stripping event handlers such as `onclick` or `onload`.
4. Filter `href` attributes so that links with a `javascript:` scheme are not preserved.
5. Return the cleaned `innerHTML` as a plain string that can be safely passed to a React component.

Together, escapeHtml and sanitizeHtml implement a defence in depth strategy. The calculators module treats strings as untrusted data and encodes them; the psychiatry right panel treats HTML fragments as untrusted structure and prunes them.

This protection can be summarised as three conceptual stages:

- **Escaping in calculators.** At the boundary between scoring logic and HTML generation, all externally supplied strings are passed through `escapeHtml`. This guarantees that patient labels, dates and flags cannot inject arbitrary tags or script content, while keeping the scoring functions themselves entirely independent of HTML details.
- **Sanitisation in the right panel.** Before any autoscore fragment reaches the live user interface, `sanitizeHtml` reduces it to a limited subset of HTML that is suitable for read only previews. Even if a future change to the calculators produced richer markup, the right panel would still enforce the same structural and attribute level constraints.
- **Controlled embedding in UI components.** Right panel components such as `RightPanelFourBlock.tsx` embed the sanitised fragment inside a single styled container that behaves like a static preview. No executable code from the fragment is ever evaluated; the report is rendered purely as content, suitable for print, export, or secure AI prompting.

Autoscore cards in the psychiatry MBC view. Autoscore reports are surfaced through dedicated cards in the psychiatry workspace rather than as stand alone utilities. The content module `src/features/psychiatry/content/psychometrics.ts` defines a set of psychometric tools. Each entry is a structured object (for example a `PsychometricItem`) with at least:

- a `kind` field that distinguishes autoscore instruments from simpler diary or checklist tools,
- a `subtype` that is aligned with `MeasureId` (for example `'phq9'`, `'gad7'`, `'pcl5'`),
- descriptive text and references used to populate the narrative view.

Right panel utilities inspect this metadata and build an example for each card. For autoscore instruments the pipeline is:

1. Determine the scale identity m from the card (for example via `measureOf(card)`).
2. Retrieve a small sample response vector $\mathbf{x}_{\text{sample}}$ and example render options o_{sample} from internal fixtures.
3. Compute a sample report

$$h_{\text{sample}} = \text{renderAutoscoreHTML}\big(m,\ \mathbf{x}_{\text{sample}},\ o_{\text{sample}}\big).$$

4. Sanitize and embed this report in the render tab of the right panel so that clinicians see an exact preview of the scale specific layout.

Thus a single autoscore card simultaneously offers: (a) a narrative description of when and why the scale is useful in MBC, (b) references to the psychometric literature and guideline context, and (c) a live, example report that demonstrates how the numeric outputs are rendered for clinical use.

Although the current examples use fixed sample vectors, the same function composition can be applied to real patient data. For any instrument m, real response vector $\mathbf{x}$, and render options o, the system can compute

$$\text{HTML}_{\text{mbc}} = \text{sanitizeHtml}\Big(\text{renderAutoscoreHTML}(m,\ \mathbf{x},\ o)\Big),$$

and embed the result in an encounter note, a discharge letter, or an export surface, while still using exactly the same scoring rules described in Section 6.2.

Figure 11: *Autoscore pipeline from item responses to sanitised HTML report in the psychiatry MBC view. Each stage is a pure function; the overall map* sanitizeHtml ∘ renderAutoscoreHTML ∘ f_m ∘ coerce *has no side effects and can be tested independently of the user interface.*

6.4 Cross-Language Transparency

A core design goal of the measurement-based care (MBC) engine is that its scoring logic should be both *auditable* and *portable* across different technical ecosystems. In practice, depression severity for the PHQ-9 is computed at runtime inside the TypeScript MBC module (`mbc/calculators.ts`), while many clinical research workflows and quality checks are still performed in Python or R notebooks. To bridge these worlds, SynapseIDE exposes paired, human-readable reference implementations of the same PHQ-9 scoring function in both TypeScript and Python via `components/ai/psychiatry/score.ts`. These snippets are presented to the clinician or informatics user as *executable documentation*: short, copyable code that makes the exact mapping from item responses to total score and severity bands explicit.

Formally, let $x = (x_1, \ldots, x_9) \in \{0,1,2,3\}^9$ denote the vector of PHQ-9 item responses. The raw total score is defined as

$$S_{\text{PHQ9}}(x) = \sum_{i=1}^{9} x_i, \qquad x_i \in \{0,1,2,3\}. \tag{6.8}$$

The corresponding severity band is obtained by applying a piecewise mapping $\text{sev} : \{0,\ldots,27\} \to \mathcal{L}$, where $\mathcal{L}$ is the finite set of verbal labels:

$$\text{sev}(s) = \begin{cases} \text{minimal}, & 0 \le s \le 4, \\ \text{mild}, & 5 \le s \le 9, \\ \text{moderate}, & 10 \le s \le 14, \\ \text{moderately severe}, & 15 \le s \le 19, \\ \text{severe}, & 20 \le s \le 27. \end{cases} \tag{6.9}$$

In the full MBC engine, these bands are represented as `ScoreBand` objects and assembled into a `ScoreResult` together with flags (e.g. "Item 9 > 0 — discuss safety today"), as described in Section 6. The cross-language transparency layer focuses on re-expressing the core transformation $x \mapsto (S_{\text{PHQ9}}(x), \text{sev}(S_{\text{PHQ9}}(x)))$ in both TypeScript and Python in a way that is concise yet mathematically identical.

Let F_{TS} and F_{PY} denote the functions implemented by the TypeScript and Python snippets, respectively. On the domain of valid PHQ-9 response vectors $D = \{0,1,2,3\}^9$, we explicitly require the following invariance property:

$$F_{\text{TS}}(x) = F_{\text{PY}}(x) \quad \text{for all } x \in D, \tag{6.10}$$

where both sides return the pair $(\text{score}, \text{severity})$. In other words, given any admissible response pattern, the total score and verbal severity label computed by the TypeScript snippet must match those computed by the Python snippet exactly. The runtime `phq9Score` function used by the MBC engine is more elaborate (it returns bands, flags and optional breakdown fields), but its *numeric* total is constrained to coincide with S_{PHQ9} in (6.8). The cross-language snippets provide a minimal, but fully aligned, view of the same rule.

Reference snippets in TypeScript and Python. The file `components/ai/psychiatry/score.ts` exports two string constants, `SCORE_PHQ9_TS` and `SCORE_PHQ9_PY`, which are wired into the psychiatry catalog and surfaced as "code" items in the AI panel. The TypeScript version is a streamlined restatement of the core scoring logic:

```
// PHQ-9 Scoring (TypeScript)

export function scorePHQ9(answers: number[]) {
  if (!Array.isArray(answers) || answers.length !== 9) {
    throw new Error('Expected 9 answers');
  }
  const invalid = answers.some(
    a => a < 0 || a > 3 || !Number.isFinite(a)
  );
  if (invalid) {
    throw new Error('Answers must be integers 0-3');
  }
  const score = answers.reduce((s, v) => s + v, 0);
  let severity: string;
  if (score <= 4) severity = 'minimal';
  else if (score <= 9) severity = 'mild';
  else if (score <= 14) severity = 'moderate';
  else if (score <= 19) severity = 'moderately severe';
  else severity = 'severe';
  return { score, severity };
}
```

The Python snippet mirrors the same invariants, written so that it can be pasted directly into a Jupyter notebook or research script:

```
# PHQ-9 Scoring (Python snippet)
# answers: list of nine integers 0-3

def score_phq9(answers: list[int]) -> dict:
    if len(answers) != 9:
        raise ValueError('Expected 9 answers')
    if any((a < 0 or a > 3) for a in answers):
        raise ValueError('Answers must be integers 0-3')
    score = sum(answers)
    if score <= 4:
        severity = 'minimal'
    elif score <= 9:
        severity = 'mild'
    elif score <= 14:
        severity = 'moderate'
    elif score <= 19:
        severity = 'moderately severe'
    else:
        severity = 'severe'
    return { 'score': score, 'severity': severity }
```

Both snippets explicitly validate the number of items and admissible range (nine answers in $[0, 3]$), compute the sum as in (6.8), and apply the same banding logic as in (6.9). The TypeScript version uses a plain object with `score` and `severity` fields, while the Python version returns a dictionary with the same keys. In the full MBC pipeline, these outputs can be compared against the richer `ScoreResult` returned by `phq9Score` to confirm that the numeric total and severity interpretation are consistent.

Documentation, portability, and quality control. By exposing these paired implementations through the psychiatry AI catalog, SynapseIDE enables three complementary uses:

- *Educational transparency.* Trainees can see, line by line, how raw Likert responses are transformed into a clinically interpreted severity level. The presence of both TypeScript and Python versions makes it clear that the logic is not tied to a particular framework or vendor.
- *Research portability.* Researchers who export de-identified response data from SynapseIDE can copy the Python snippet into their statistical environment and recompute PHQ-9 scores locally, without having to re-derive the scoring rules or trust a black-box export.
- *Verification and regression checks.* Because both snippets implement the same function F_{PHQ9} on the domain D, they can be used in test harnesses to detect regressions: any change to the MBC engine's implementation has to preserve equation (6.10) for all admissible inputs.

Table 24 summarises the main invariants that are guaranteed to hold across the TypeScript and Python reference implementations. This table complements the band definitions in Section 6 by making explicit which parts of the PHQ-9 logic are intended to be language-independent.

Table 24: *Key invariants enforced between the TypeScript and Python PHQ-9 reference implementations. The domain is restricted to valid response patterns $D = \{0, 1, 2, 3\}^9$.*

Invariant	Implementation artefacts	Description
Domain validation	`scorePHQ9` (TS); `score_phq9` (Py)	Both functions require exactly nine answers and accept only values in the range 0–3; invalid inputs raise an error rather than being silently coerced.
Total score equality	`score` field (TS/Py)	For any $x \in D$, both implementations compute `score` $= S_{\text{PHQ9}}(x)$ as in equation (6.8).
Severity band equality	`severity` field (TS/Py)	For any admissible total s, the severity label returned by both snippets corresponds to $\text{sev}(s)$ in equation (6.9), ensuring identical banding thresholds across languages.
Determinism	Pure functions without side effects	Given the same input vector x, both functions return the same {`score`, `severity`} object on every call; there is no dependence on external state or randomness.

6.5 Future Longitudinal Analytics

The current measurement-based care (MBC) engine focuses on cross-sectional scoring: each administration of PHQ-9, GAD-7, PCL-5, Y-BOCS, or AUDIT-C is converted into a `ScoreResult` with bands and flags (Section 6). In routine clinical work, however, the most clinically meaningful signal lies in the trajectory of scores over time and in how different

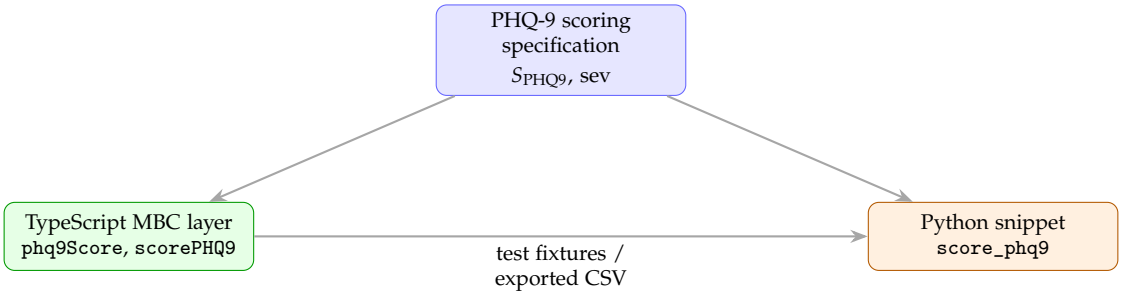


Figure 12: *Cross-language transparency pattern for PHQ-9: a single mathematical specification is implemented in the TypeScript MBC engine and mirrored as a Python snippet, allowing external validation and research re-use without re-deriving the scoring rules.*

symptom domains move together. This subsection specifies the planned longitudinal analytics layer that will be implemented on top of the existing SynapseCore data model in later chapters.

At the code level, each patient in `src/centerpanel/registry/types.ts` is already equipped with:

- a risk grade `risk: RiskLevel` (with levels 1–5),
- optional cross-sectional fields such as `phq9Score` and `phq9Delta`,
- arrays of structured `assessments[]` and `encounters[]` with timestamps `when`,
- optional `snapshots[]` capturing concise encounter summaries at specific time points.

Future longitudinal analytics will treat these as building blocks for time-indexed streams, cross-scale composite indices, and discrete risk grade trajectories.

Time-indexed score streams. Let $p \in \mathcal{P}$ denote a patient and $\mathcal{K} = \{\text{PHQ9}, \text{GAD7}, \text{PCL5}, \dots\}$ the set of supported scales. For each scale $k \in \mathcal{K}$, the MBC calculators already provide a total-score map

$$S^{(k)} : \mathbb{Z}_{\geq 0}^{n_k} \to \{0, \dots, M_k\},$$

where n_k is the number of items in scale k and M_k is its maximum possible total (e.g. $M_{\text{PHQ9}} = 27$, $M_{\text{GAD7}} = 21$, $M_{\text{PCL5}} = 80$).

For patient p, let $t = 1, \dots, T_p$ index successive encounters or measurement time points, ordered by the timestamp field `when` in the `Encounter` or `Assessment` record. The planned longitudinal state for scale k is the discrete time series

$$S_{p,1}^{(k)},\ S_{p,2}^{(k)},\ \dots,\ S_{p,T_p}^{(k)}, \qquad S_{p,t}^{(k)} \in \{0, \dots, M_k\}, \tag{6.11}$$

obtained by applying the existing calculators (e.g. `phq9Score`) to each recorded administration of the corresponding instrument.

Implementation-wise, this corresponds to a lightweight adapter that:

1. iterates over `assessments[]` and/or `encounters[]`,
2. reconstructs per-scale totals from stored answers or cached `ScoreResult.total` values,
3. emits an ordered list of points $(t, S_{p,t}^{(k)})$ per scale.

Irregular spacing in calendar time is preserved through the underlying `when` field (a millisecond timestamp), allowing later chapters to define both discrete-time indices t and continuous-time indices τ (e.g. days since baseline) for change metrics and survival-like analyses.

Cross-scale composite indices. Many future use-cases require a single summary index that captures the combined burden of symptoms across several domains (e.g. depression, anxiety, post-traumatic stress). To that end, we define a composite index $C_{p,t}$ built from normalised scale scores.

For each scale $k \in \mathcal{K}$, define the normalised score

$$\tilde{S}_{p,t}^{(k)} = \frac{S_{p,t}^{(k)}}{M_k} \in [0,1], \tag{6.12}$$

and choose non-negative weights $w_k \geq 0$ with $\sum_{k\in\mathcal{K}} w_k = 1$. The composite longitudinal index for patient p at time t is then

$$C_{p,t} = \sum_{k\in\mathcal{K}} w_k \, \tilde{S}_{p,t}^{(k)}. \tag{6.13}$$

By construction, $C_{p,t} \in [0,1]$, where 0 represents no reported symptoms on any contributing scale, and 1 corresponds to the maximum possible severity across all selected scales.

In the codebase, this composite index will be exposed as a small pure function,

```
type CompositeIndex = {
  value: number;  // 0-1
  components: Record<string, number>; // per-scale contributions
};

function computeCompositeIndex(p: Patient, at: number): CompositeIndex { ... }
```

which takes a snapshot of the patient's scores at a given time (derived from `assessments[]` and the MBC calculators), applies equation (6.12) and weights w_k, and returns both the scalar composite value and its per-scale components. Later chapters will specify how w_k are chosen (e.g. clinician-defined, data-driven, or hybrid schemes).

Risk grade mapping and trajectories. The SynapseCore registry already distinguishes five risk grades via the `RiskLevel` type and the helper in `src/centerpanel/rail/riskGrades.ts`, which maps grades to UI labels such as "G1 • Low" or "G4 • High". The longitudinal layer extends this static view into a time-indexed risk trajectory.

Let

$$\mathbf{S}_{p,t} = \left(S_{p,t}^{(\mathrm{PHQ9})}, S_{p,t}^{(\mathrm{GAD7})}, S_{p,t}^{(\mathrm{PCL5})}, \dots\right)^{\top}$$

denote the vector of per-scale totals at time t for patient p. The conceptual risk mapping is

$$R : \mathbb{R}^d \to \{1,2,3,4,5\}, \qquad R\left(\mathbf{S}_{p,t}\right) = G_{p,t}, \tag{6.14}$$

where d is the number of scales considered and $G_{p,t}$ is the risk grade at time t. In the simplest version, R is a threshold rule applied to the composite index $C_{p,t}$ in equation (6.13):

$$G_{p,t} = \begin{cases} 1, & 0 \leq C_{p,t} \leq q_1, \\ 2, & q_1 < C_{p,t} \leq q_2, \\ 3, & q_2 < C_{p,t} \leq q_3, \\ 4, & q_3 < C_{p,t} \leq q_4, \\ 5, & q_4 < C_{p,t} \leq 1, \end{cases} \tag{6.15}$$

with $0 < q_1 < q_2 < q_3 < q_4 < 1$ chosen to reflect clinically interpretable transitions (e.g. from low to moderate or from moderate to high acute risk).

More advanced variants, to be detailed in later chapters, can replace the simple thresholding in (6.15) by a learned mapping R_θ (e.g. logistic regression or gradient boosting) trained on historical outcome data while still emitting discrete grades 1–5 that are compatible with the existing `RiskLevel` type and UI components.

In longitudinal form, each patient will have a risk grade sequence

$$G_{p,1}, G_{p,2}, \ldots, G_{p,T_p}, \tag{6.16}$$

aligned with encounters via their `when` timestamps. This sequence can be used for trajectory plots, change detection (e.g. flags for rapid upgrades from G1 to G4), and cohort-level metrics such as the proportion of patients with a sustained downgrading over a fixed follow-up horizon.

Planned implementation hooks and layers. Table 25 summarises the main layers of the planned longitudinal analytics stack and how each layer relates to the current SynapseCore code.

Table 25: *Planned longitudinal analytics layers and their alignment with existing SynapseCore data structures. Column widths are fixed to avoid page overflow.*

Analytics layer	Mathematical object	Planned implementation hook
Per-scale trajectories	Per-patient sequences $S_{p,t}^{(k)}$ as in (6.11)	Derived from `assessments[]` and `ScoreResult.total` values, ordered by encounter timestamps `when`.
Cross-scale composites	Normalised and weighted index $C_{p,t} \in [0,1]$ as in (6.13)	Implemented as a pure `computeCompositeIndex` helper that aggregates per-scale totals into a single scalar with component breakdown.
Risk grade trajectories	Discrete sequence $G_{p,t} \in \{1,\ldots,5\}$ as in (6.16)	Uses the existing `RiskLevel` type and `getRiskGradeInfo` mapping; future functions compute grade histories and flag rapid upgrades or downgrades.
Cohort-level summaries	Distributions and change statistics over $\{C_{p,t}\}$ and $\{G_{p,t}\}$	Exposed through registry filters, the Clinical Snapshot rail, and planned extensions to the ML Insights panel for aggregated views over selected cohorts.

Conceptual longitudinal schema. Figure 13 gives a compact schema linking per-encounter MBC scores to composite indices and risk grades over time. It is designed as an outline: later chapters will refine each arrow into concrete algorithms and evaluation criteria.

7 Structured Clinical Flows

Structured clinical flows in SynapseCore encode recurrent, high–stakes psychiatric situations as guided, stateful documentation wizards. The aim is to make critical workflows (e.g., acute safety reviews, capacity checks, agitation episodes) reproducible and auditable while keeping the interaction load on the clinician low. Conceptually, each flow is a typed finite–state

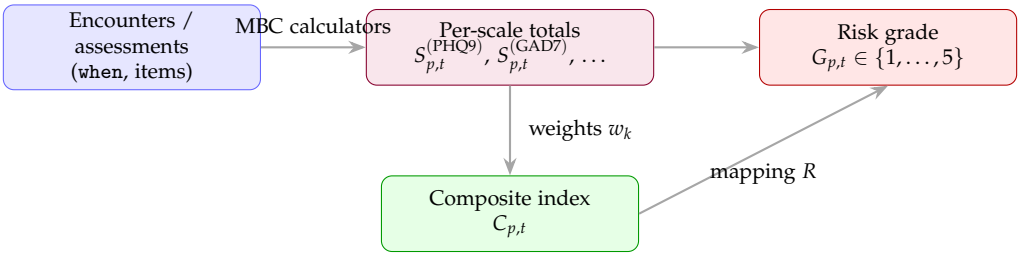


Figure 13: *Outline of the planned longitudinal analytics pipeline: each encounter with completed scales feeds the existing MBC calculators to produce per-scale totals, which are combined into a composite index $C_{p,t}$, and then mapped to a discrete risk grade $G_{p,t}$.*

machine that accumulates structured form data and then converts it into a neutral narrative snippet that can be attached to the current encounter and reused as AI context or in exports.

At implementation level, the flows framework lives under `src/centerpanel/Flows/`, with layout and typography governed by `src/centerpanel/styles/flows.module.css`. The core responsibilities are split across:

- orchestration of which flow or view is currently active;
- presentation of the flow library (what flows exist, for which use case);
- shared layout, step navigation, and legal boundary copy;
- glue into the encounter/registry so completed runs can be stored and reviewed.

7.1 Flows Framework Overview

The central orchestrator, `FlowHost.tsx`, is responsible for deciding *what* to render in the flows panel at a given time. It imports:

- the individual flow shells (e.g., `SafetyFlowShell`, `AgitationFlowShell`, `BFCRSShell`, `LorazepamChallengeShell`, `CapacityFlowShell`, `ObservationFlowShell`);
- the read–only viewer `ReadOnlyRunView`;
- the UI store `useFlowsUIStore` and the flow identifier type `FlowId`;
- global legal copy (`GLOBAL_FLOW_BOUNDARY_LINE`, `GLOBAL_FLOW_SUBTITLE`).

Internally, `FlowHost.tsx` tracks a small view state:

- current mode (e.g. active flow versus completed–run review);
- active `flowId` or `null` when the panel is idle;
- the identifier of the completed run being inspected in review mode.

This state is held in `uiStore.ts` using a lightweight Zustand store, so that the rest of the CenterPanel can read whether a flow is currently active without having to know about individual shells.

From a formal perspective, the host and its store implement a global flow controller

$$C = \big(\mathcal{F}, f_{\text{active}}, m, \Theta\big),$$

where $\mathcal{F}$ is the finite set of available flows (keys of `FlowId`), $f_{\text{active}} \in \mathcal{F} \cup \{\varnothing\}$ is the currently active flow (or none), m is the coarse view mode (e.g. `"flow"` vs. `"review"`), and Θ is the set of completed–run selectors. UI events (clicking a library tile, opening a completed run, exiting review) induce transitions

$$T_C : (f_{\text{active}}, m, \Theta) \longrightarrow (f'_{\text{active}}, m', \Theta'),$$

implemented as tiny store actions in `useFlowsUIStore`.

Flow library and metadata. The flow library presentation layer is implemented by `FlowLibraryCard.tsx` together with the metadata registry `flowLibraryMeta.ts`. The registry exports a constant such as `FLOW_LIBRARY_ITEMS`, which is an array of objects describing each flow:

- a stable `flowId` (e.g. `"safety"`, `"agitation"`, `"capacity"`, `"catatonia"`);
- title and category labels used in the UI;
- a short "clinical focus" sentence;
- a list of what is documented in the flow (e.g. ideation, plan, access to means for safety);
- a per–flow boundary line which reinforces that the flow is for documentation and does not, on its own, constitute orders or legal determinations.

`FlowLibraryCard.tsx` uses this metadata to render a small, two–column grid of tiles. Each tile is implemented by `FlowTile.tsx` and styled via `flows.module.css`, exposing:

- the flow title and clinical focus;
- an optional badge for flows that are locked or in a future release;
- the global boundary disclaimer anchored at the bottom of the card.

Clicking a tile dispatches an action into `useFlowsUIStore` to set `flowId` and enter `"flow"` mode. `FlowHost.tsx` then selects the corresponding shell component (e.g. `SafetyFlowShell`) in a simple switch statement, keeping the routing logic in one place.

Layout and styling. All flows share a common layout defined in `centerpanel/styles/flows.module.css`. The stylesheet provides classes for:

- the outer container (root), which uses a flex or grid layout tuned for the CenterPanel width;
- header rows with title, subtitle, and a compact step indicator ("Step k of K");
- the main grid of labels and form fields, constrained with `max-width: 860px` to prevent horizontal overflow in the article's A4 layout;
- step pills (`StepPills.tsx`) and footer controls (*Back*, *Next*, *Complete*).

By keeping these styles in a dedicated CSS module, the flows framework can present very different clinical processes with a uniform visual language: monospace micro–labels, consistent padding, and buttons that match the psychiatry toolkit and timer components.

Finite–state view of a single flow. While `FlowHost.tsx` operates at the level of *which* flow is active, each individual flow shell behaves like a small finite–state machine. Abstractly, for a given flow F we can define:

$$F = \big(S, s_0, E, T, \rho\big),$$

where:

- S is the finite set of steps (UI screens),
- $s_0 \in S$ is the initial step,
- E is the set of user events ("next", "back", field edits),
- $T : S \times E \to S$ is the transition function implementing step navigation,
- ρ is a data accumulator that folds all field edits into a typed form object (e.g. `AgitationFormState`).

At completion time, the flow passes the accumulated form object into an outcome builder (implemented in `Flows/builders/*.ts`) which returns both a full narrative paragraph and a preview string. These outputs are stored in the encounter registry as a *completed run* and can later be viewed by `ReadOnlyRunView`.

Table 26: *Core elements of the structured flows framework in* `centerpanel/Flows/`*. Column widths are chosen to avoid page overflow.*

Artifact	Responsibility	Illustrative contents
`FlowHost.tsx`	Orchestration and routing	Selects which flow shell or read–only view to render; reads and writes `flowId`, mode, and selected run from `useFlowsUIStore`.
`FlowLibraryCard.tsx` + `flowLibraryMeta.ts`	Library UI and metadata	Registry of `FlowId`s with titles, categories, clinical focus and boundary text; renders tiles with shared legal copy.
`flows.module.css`	Layout and styling	Container layout, headers, step pills, form grid, and buttons; uses a capped `max-width` to prevent horizontal scrolling.
`FlowTile.tsx`, `StepPills.tsx`, `ReadOnlyRunView.tsx`	Shared UI components	Flow selection tiles, step navigation controls, and read–only display of completed runs.
`uiStore.ts`, `flowTypes.ts`	State and typing	Zustand store for view state; `FlowId` and related TypeScript types shared by shells, builders, and the registry.

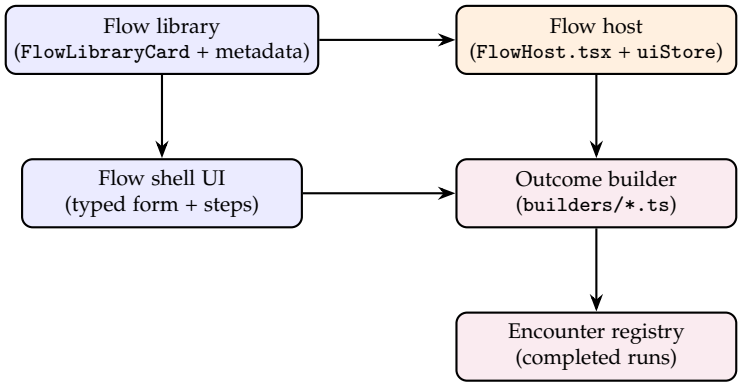


Figure 14: *High–level stack of the flows framework. The coloured, two–column layout groups UI/library components on the left and orchestration/registry components on the right while remaining within the A4 text block without horizontal overflow.*

Table 26 summarises the main framework artifacts without going into individual clinical flows.

7.2 Domain Flows

Beyond the shared framework described in Section 7.1, SynapseCore ships a set of concrete domain flows implemented as typed form states and outcome builders. Each flow is represented by a pair (X_F, b_F), where X_F is the TypeScript form state for that flow and

$$b_F : X_F \times \mathbb{R}_{\geq 0} \longrightarrow \Pi$$

is a builder function mapping a completed form state and an insertion timestamp (measured in milliseconds) to a neutral narrative paragraph $\pi \in \Pi$ suitable for the encounter record and downstream AI prompts.

All builders live under `centerpanel/Flows/builders/`, with filenames of the form

Table 27: Structured clinical flows and their associated form states and builder functions. Column widths are chosen to avoid page overflow.

Flow	Form state / builder	Primary documentation focus
Agitation / de-escalation	`AgitationFormState` `agitationOutcome.ts`	Objective behaviour, injury risk profile, de-escalation techniques, escalation rationale, post-intervention status, reassessment and handoff.
Capacity / consent	`CapacityFormState` `capacityOutcome.ts`	Decision context plus understanding, appreciation, reasoning and expressed choice, with a neutral capacity impression and follow-up plan.
Catatonia (BFCRS-aligned)	`CatatoniaFormState` `catatoniaOutcome.ts`	Observed catatonic signs, BFCRS-aligned severity, functional impact, risk factors, monitoring and handoff.
Lorazepam challenge	`LorazepamFormState` `lorazepamOutcome.ts`	Indication context, pre-dose status, dose/route, immediate response, safety observations, monitoring and reassessment needs.
Observation / containment	`ObservationFormState` `observationOutcome.ts`	Behavioural narrative, immediate safety risk type, de-escalation attempts, observation level and rationale, monitoring and handoff.
Safety / suicide risk review	`SafetyFormState` `safetyOutcome.ts`	Ideation and intent status, access to means, acute risk and protective factors, willingness to alert staff, and observation discussion.

`*Outcome.ts`. Typed form states are defined in `centerpanel/Flows/types/`. Table 27 summarises the bundled flows.

Each builder enforces three properties that are important for clinical and medicolegal safety:

1. **Time anchoring.** All flows call a timestamp helper (e.g. `formatLocalTimestamp` or `tsLocal`) so that the resulting paragraph explicitly states when the assessment was recorded.
2. **Deterministic language.** Enumerated codes in the form state are mapped to fixed phrases via small helper functions (e.g. `describeInjuryRisk`, `mapUnderstanding`, `riskTypeToHuman`). This ensures that similar clinical states produce consistent wording across encounters.
3. **Neutral, non-coercive framing.** Builders include boundary sentences that emphasise documentation, supervision and communication, and explicitly state that the paragraph is not, by itself, a treatment order, seclusion/restraint authorisation or legal determination.

Agitation and de-escalation. The agitation flow uses `AgitationFormState` to capture the full arc of a behavioural emergency:

- *Objective behaviour* (`objectiveBehavior`) records what is visible (e.g. pacing, shouting, throwing objects) without interpretation.
- *Injury risk profile* (`injuryRiskProfile`) categorises observed or attempted harm (verbal only, attempted self-harm, attempted assault, property damage, not assessed). The builder's `describeInjuryRisk` function maps these codes to short, neutral phrases.

- *Medical contributors* (`medicalContributorsConsidered`) documents potential medical or pharmacologic drivers (e.g. delirium, intoxication, withdrawal).
- *De-escalation techniques* are represented by a nested object `deescalationTechniques` with boolean fields such as `verbalDeescalation`, `calmEnvironment`, `reducedStimuli`, `offeredSupportivePresence` and `setClearRespectfulLimits`. A helper routine collects all `true` flags into a list of human-readable items.
- *Response to de-escalation* (`responseToDeescalation`) is coded as calmed, partially calmed, no effect or escalated.
- *Escalation and least-restrictive rationale* fields capture what escalation was discussed or undertaken, how less restrictive options were considered insufficient, and the reasoning behind any escalation.
- *Post-intervention status, reassessment plan and staff notified* complete the time course and clarify supervision.

The agitation builder `agitationOutcome.ts` assembles these elements into a paragraph that narrates, in order: initial objective behaviour, risk profile, non-pharmacologic de-escalation attempts and response, escalation (if any) with explicit reference to least-restrictive efforts, and the post-intervention plan. Missing or blank fields are normalised to `"[not recorded]"` to avoid implicit omissions.

Capacity and consent. The capacity flow formalises the classical four-pillar model of decision-specific capacity. `CapacityFormState` includes:

- *Decision context* (`decisionContext`), grounding the assessment in a specific choice (e.g. leaving hospital, starting a medication).
- Level and evidence fields for *understanding*, *appreciation* and *reasoning* (e.g. `understandingLevel` with a corresponding `understandingVerbatim`).
- *Expressed choice* and *choice stability* fields, plus free text notes.
- A *clinician capacity impression* and *follow-up plan* describing supervision, second opinions or re-evaluation.

Helper functions such as `mapUnderstanding`, `mapAppreciation` and `mapReasoning` convert discrete levels (e.g. adequate, partial, inaccurate, not assessed) into fixed sentences. `capacityOutcome.ts` then combines these mapped phrases with the decision context and follow-up plan into a single impression, explicitly stating that the note supports clinical communication and does not substitute for legal adjudication of capacity.

Catatonia and BFCRS-aligned documentation. `CatatoniaFormState` encodes a checklist of catatonic features in an `observedFeatures` object (mutism, stupor/immobility, posturing or catalepsy, rigidity, negativism, echolalia/echopraxia, waxy flexibility, fixed staring, withdrawal/refusal of intake, autonomic instability) plus narrative fields for observed behaviour, BFCRS-aligned severity, functional impact, risk factors, monitoring plan and handoff communication.

The builder `catatoniaOutcome.ts` uses the helper `listObservedFeatures` to convert the boolean feature map into an ordered list of human-readable descriptors, translates severity codes into short explanatory phrases, and assembles a paragraph that links severity to functional impact and concrete monitoring steps.

Lorazepam challenge. The Lorazepam challenge flow is modelled by `LorazepamFormState`, which includes indication context, pre-dose mental status and motor findings, baseline air-

way/vitals, dose and route details, immediate response, safety observations, follow-up monitoring plan and reassessment needs.

`lorazepamOutcome.ts` anchors the paragraph with a local timestamp, then narrates:

1. indication and diagnostic context (e.g. suspected catatonia);
2. pre-dose baseline mental, motor and physiological status;
3. challenge dose/route/time;
4. immediate clinical response (improvement, no change, adverse effect);
5. safety monitoring and follow-up plan.

As in other flows, unfilled fields are normalised, and the closing sentence reiterates that the documentation supports diagnostic reasoning and safety monitoring, not standing medication orders or restraint authorisation.

Observation, containment and safety review. The observation/containment flow uses `ObservationFormState` to record:

- a behaviour narrative (`behaviorNarrative`);
- the type of *immediate safety risk* (`imminentRiskType`) and a clarifying phrase;
- prior de-escalation attempts and comfort measures;
- the chosen observation level, rationale and monitoring details;
- reassessment frequency, de-escalation criteria and handoff plan.

`observationOutcome.ts` applies small mapping functions such as `riskTypeToHuman` and `observationLevelToHuman` to render risk and observation codes into concise, safety-focused language and emphasises that observation levels are framed around patient safety rather than punishment.

Finally, the safety review flow (implemented with `SafetyFormState` and `safetyOutcome.ts`) documents suicidal/self-harm ideation status, intent/plan, access to means, acute risk and protective factors, willingness to alert staff and whether observation or monitoring strategies were discussed. The builder constructs a short series of sentences, each focusing on one of these domains, and closes with a standard boundary statement clarifying that the note supports safety planning and communication rather than determining legal status or replacing local policies.

7.3 Shell Components and Interaction

The domain flows described in Section 7.2 are implemented as small, self-contained React "shell" components under `centerpanel/Flows/shells/`. Each shell owns its local form state and step index, renders a multi-step layout, and delegates final narrative construction to the corresponding outcome builder. Despite targeting different clinical problems, the shells follow a common interaction pattern so that clinicians experience a uniform workflow.

Generic shell pattern. Each flow shell (e.g. `AgitationFlowShell.tsx`, `SafetyFlowShell.tsx`, `CapacityFlowShell.tsx`, `CatatoniaFlowShell.tsx`, `LorazepamChallengeShell.tsx`, `ObservationFlowShell.tsx`)declares a small, fixed list of steps:

```
const STEPS = [
  { key: "ideation",   label: "Ideation & Intent" },
```

```
  { key: "means",      label: "Means & Acute Risk" },
  { key: "protective", label: "Protective Factors" },
  { key: "plan",       label: "Observation & Plan" },
] as const;
```

and maintains a step index and typed form state via `useState`:

```
type StepIndex = 0 | 1 | 2 | 3;
const [step, setStep] = useState<StepIndex>(0);
const [form, setForm] = useState<SafetyFormState>(defaultSafetyFormState);
```

Navigation uses a simple clamping helper

$$\text{clamp}(n, 0, K-1) = \max\{0, \min\{K-1, n\}\},$$

so that "next" and "back" never leave the valid step range:

$$s' = \text{clamp}(s + \Delta, 0, K-1), \qquad \Delta \in \{-1, +1\}.$$

Visually, each shell is organised into four layers:

1. **Header.** A header block (`flowHeader`) contains the flow title, a step progress label such as `"Step 2 of 4"`, the global subtitle (`GLOBAL_FLOW_SUBTITLE`), a warning block with flow-specific safety guidance (e.g. `SAFETY_WARN` or `AGITATION_WARN`), and a shared boundary line (`GLOBAL_FLOW_BOUNDARY_LINE`).
2. **Step navigation.** Immediately below the header, the shell renders `<StepPills />` with the current index and step labels. This component uses ARIA tab semantics: `role="tablist"` on the container and `role="tab"` on each pill, with keyboard support for left/right arrows, `Home` and `End`. Internally, StepPills maintains a ref array of buttons and a `focusIndex` helper so that keyboard navigation moves both the active index and DOM focus.
3. **Step content.** The main body (`flowBodyArea`) shows exactly one `stepContentCard` at a time, via conditions such as `step === 0`, `step === 1`, etc. Each card contains a columnar grid of `formSections`, using labels, textareas, short text inputs, boolean button groups and hints. All cards share typography, spacing and width constraints defined in `flows.module.css`, so even multi-line, high-density steps stay within an `max-width` tuned for the CenterPanel main column.
4. **Summary and insertion controls.** The final part of each shell exposes an "Insert outcome to Note" button (class `insertOutcomeButton`) plus a short explanatory hint. Where insertion is implemented (e.g. safety, capacity), the button invokes a local handler (`handleInsertOutcome` or `handleInsert`) that calls the appropriate builder, appends the resulting paragraph to the current encounter note via the registry actions, and logs the new run under the encounter's `completedRuns` list. In flows that are still documentation-only, the same button is rendered with the `disabled` attribute set, together with hint text noting that the insertion behaviour is not yet active.

Validation and completion states. Rather than a monolithic validation system, flows use a combination of typed state, local guards and insertion gating:

- **Typed form state.** Each flow shell is parameterised by a strongly typed form state (e.g. `AgitationFormState`, `CapacityFormState`), so the UI cannot produce structurally invalid objects: missing fields or wrong types are caught at build time.

- **Local guards.** In the capacity shell, for example, the insertion button is enabled only when there is an active patient and encounter:

```
const hasActiveSelection =
  Boolean(state.selectedPatientId && state.selectedEncounterId);
...
<button
  className={styles.insertOutcomeButton}
  onClick={handleInsert}
  disabled={!hasActiveSelection}
>
  Insert outcome to Note
</button>
```

  This pattern prevents detached flow runs from being generated without a concrete clinical context.
- **Completion timestamps.** Shells record the last insertion time (e.g. `lastInsertedAtMs`) so that repeated clicks can be handled explicitly and downstream audit trails can reconstruct when the summary was appended.
- **Per-step expectations.** Some shells guide completion through hint text rather than hard blocking (for example, suggesting that protective factors or least-restrictive rationales be filled before moving on). In future iterations, these expectations can be made explicit as validation functions $v_F : X_F \to \{\text{valid}, \text{invalid}\}$ that determine whether the final insertion is allowed.

Integration with the CenterPanel shell. The flows UI is integrated into the main shell via the `"Flows"` tab of `CenterPanelShell.tsx`. Tabs are defined as

```
type Tab = "Registry" | "New Patient" | "Guide" | "Note" | "Flows" | "Tools";
const TABS: Tab[] = ["Registry", "New Patient", "Guide", "Note", "Flows", "Tools"];
```

When `activeTab === "Flows"`, the layout mirrors the other psychiatry views:

- The left rail is rendered as

```
<nav
  key={activeTab}
  className={`${styles.outline} noPrint ${styles.panelEnter}`}
  aria-label="Flows left rail"
  data-testid="cp-outline"
>
  <FlowsRail ... />
</nav>
```

  where `FlowsRail` combines the flow library (`FlowLibraryCard`), suggested flows and a "Completed This Encounter" card using the same `outline` width and vertical rhythm as the Note and Tools rails.
- The main content column is rendered as

```
<main id={MAIN_SCROLL_ROOT_ID}
      className={styles.main}
      role="main"
      data-testid="cp-main">
```

```
      <section
        key={activeTab}
        className={styles.panelEnter}
        id="panel-flows"
        role="tabpanel"
        aria-labelledby="tab-flows"
      >
        <FlowHost
          activeFlowId={activeFlowId}
          activeReviewRun={activeReviewRun}
        />
      </section>
    </main>
```

so that flows inherit the same scroll root, enter animation (`panelEnter`) and semantic labelling as the other CenterPanel panels.

- `FlowHost` consumes `activeFlowId` and `activeReviewRun` from `CenterPanelShell` and the `useFlowsUIStore`, switching between active shells and the `CompletedRunReviewShell` in a single place. This keeps the tab integration thin: the shell only needs to know which flow is active and whether a completed run is being reviewed; the per-flow interaction logic remains local to the shells.

Through this arrangement, structured clinical flows occupy the same visual grid and hierarchy as the registry, note and AI tools: a left rail for navigation and context, a main column for dense content, and a shared timer and header above, all wrapped by `CenterPanelShell` and the `TimerProvider`.

7.4 Flow $\rightarrow$ AI Integration

The flows framework is designed so that every completed run can be handed to the AI orchestration layer as a structured artefact rather than an opaque free-text blob. In particular, a completed flow run produces (i) a typed React form state, (ii) a deterministic narrative paragraph in natural language, and (iii) a compact, JSON-serialisable record that can be embedded into AI contexts, exported, or analysed longitudinally.

Let F denote a given flow (e.g. safety review, agitation episode, observation check). As introduced in Section 7.2, each flow is characterised by a form state space X_F and a builder

$$b_F : X_F \times \mathbb{R}_{\geq 0} \longrightarrow \Pi,$$

mapping a completed state and insertion timestamp to a paragraph $\pi \in \Pi$. The flow $\rightarrow$ AI pipeline refines this picture by introducing two additional layers:

$$X_F \xrightarrow{\sigma_F} S_F \xrightarrow{\mathrm{enc}_F} J_F \xrightarrow{\Gamma_F} C_F, \tag{7.1}$$

where

- S_F is a compact, type-safe summary of $x_F \in X_F$ in terms of clinically meaningful fields (e.g. risk codes, de-escalation measures, post-status, handoff recipients).
- $\sigma_F : X_F \rightarrow S_F$ is a deterministic summariser mapping the raw form state to S_F.[2]
- $\mathrm{enc}_F : S_F \times \Pi \rightarrow \mathcal{J}_F$ encodes $(\sigma_F(x_F), b_F(x_F, t))$ into a JSON-serialisable object $J_F \in \mathcal{J}_F$ containing both structured fields and the neutral narrative paragraph.

[2]In practice, σ_F is implemented as a small adapter that picks a stable subset of fields from the TypeScript state and rewrites them into human-readable categorical codes.

Table 28: *Stages in the flow → AI pipeline for completed runs. Column widths are chosen to avoid page overflow.*

Stage	Representation	Agitation example
1. Form capture	React state object x_F (e.g. `AgitationFormState`)	Clinician records observable behaviour, injury risk profile, suspected medical contributors, de-escalation techniques, escalation type, post-intervention status, and handoff details.
2. Outcome builder	Narrative paragraph $\pi = b_F(x_F, t)$	`buildAgitationOutcome(`x_F`, t)` produces a neutral paragraph summarising the episode; it is appended to the encounter note and stored as part of the flow run metadata.
3. Structured summary	Typed summary $s_F = \sigma_F(x_F)$	A small adapter rewrites categorical fields such as `injuryRiskProfile`, `deescalationTechniques`, `escalationTypeDiscussed`, `postInterventionStatus` into short, standardised descriptors.
4. JSON encoding	JSON object $J_F = \mathrm{enc}_F(s_F, \pi)$	The summary s_F and paragraph π are combined into a normalised JSON record with explicit keys, timestamps, and flow identifiers.
5. AI context build	AI-ready context $C_F = \Gamma_F(J_F)$	The JSON record is serialised into a context block that also includes an MSE scaffold and brief instructions about tone, length, and scope.
6. LLM response	Streamed tokens from provider	The AI engine sends C_F and a task prompt (e.g. "Draft a short narrative of the agitation episode and a risk/plan summary"); the streamed output appears in the AI panel next to the original flow data.

- $\Gamma_F : \mathcal{J}_F \to \mathcal{C}_F$ builds an AI-ready context block C_F by combining J_F with psychiatry-specific prompts (e.g. the MSE scaffold) and a short, task-specific instruction header.

The context C_F is then passed to the generic AI orchestration engine (Section 4), which maps it into a provider-specific request, streams the response, and returns generated text that can be inserted into the note or used for clinical communication.

7.4.1 Formal view and type discipline

At type level, the form state x_F is a rich object that contains both UI-level details (transient selections, free-text fragments, default values) and clinically essential content. The summariser σ_F is deliberately kept small and composable so that:

$$S_F \subseteq \prod_{k=1}^{K} \mathcal{V}_k, \tag{7.2}$$

where each $\mathcal{V}_k$ is either a finite categorical set (e.g. risk levels), a bounded integer interval (e.g. counts, Likert-style responses), or a short free-text field with explicit length constraints. This yields a structured representation that:

- is straightforward to serialise into JSON;
- can be analysed numerically or categorically across encounters; and
- can be re-used as a feature vector in downstream models without needing to re-parse free text.

The encoding step enc_F then attaches the narrative output $\pi = b_F(x_F, t)$ to s_F:

$$J_F = \mathrm{enc}_F\big(\sigma_F(x_F), b_F(x_F, t)\big) = \big(\mathrm{id}, \mathrm{timestamp}, \mathrm{flowId}, s_F, \pi\big),$$

where id is a run identifier, flowId is the stable flow key, and timestamp records the insertion time. This record is stored in the encounter object (e.g. under `completedRuns`) and can also be exported through the Tools tab.

In the current implementation, this structure is realised via:

- registry actions such as `appendFlowOutcomeAtSelection`, which populate `enc.noteSlots.outcome` and append objects of the form `{runId, flowId, label, insertedAt, paragraph, paragraphFull, paragraphPreview}` to `enc.completedRuns`;
- the export assembly pipeline in `centerpanel/Tools/lib/assemble.ts`, which uses `completedRuns` and `noteSlots` to render de-identified HTML exports; and
- the psychiatry AI catalog in `components/ai/psychiatry/packs.ts`, which collects MSE and other scaffolds into a structured library of prompts.

The same JSON-serialisable structures can be fed into AI contexts without any additional parsing.

7.4.2 JSON normalisation and AI contexts

Although flows were designed primarily for human-readable documentation, the JSON representation J_F plays a central role in AI integration. For each flow F, the context builder Γ_F is responsible for turning J_F into a string C_F that respects the following constraints:

1. *Determinism.* For fixed J_F, the context string C_F is deterministic and independent of the choice of AI model.
2. *Separation of roles.* Clinical facts and risk-relevant content live in J_F and π; the LLM is used only for language generation and formatting.
3. *Bounded size.* The serialisation respects a token budget specified by the AI engine (e.g. via a maximum context length in the runtime configuration).
4. *Scope control.* The context explicitly states that the output is for documentation and communication, not real-time triage, diagnosis, or medication authorisation.

Concretely, Γ_F constructs C_F by concatenating three components:

$$C_F = H_F \,\|\, \mathrm{serialize}\big(J_F\big) \,\|\, P_{\mathrm{MSE}}, \tag{7.3}$$

where

- H_F is a short header specifying the clinical setting and the requested output genre (e.g. "ED psychiatry consult: agitation episode documentation").
- $\mathrm{serialize}(J_F)$ prints the JSON in a compact, stable order with human-readable labels for categorical codes.
- P_{MSE} is the mental status examination scaffold from `components/ai/psychiatry/prompts.ts`, encouraging the model to organise the narrative according to familiar psychiatric headings.

Table 29: *Representative flows and their primary AI-facing tasks. The column widths are chosen to remain within the text block.*

Flow	Primary AI tasks	Example artefacts
Safety review	Summarise suicidal/self-harm risk, protective factors, and observation plan using structured fields and short narrative.	Safety/risk paragraph for the note; handoff blurb to nursing; exportable risk summary snippet.
Agitation episode	Generate a concise episode narrative and plan from structured agitation fields and the built outcome paragraph.	Episode description, de-escalation summary, and plan bullets suitable for ED notes or inpatient progress notes.
Capacity check	Organise the four abilities (understanding, appreciation, reasoning, expression of choice) into a structured, neutral summary.	Capacity documentation block that can be re-used in legal forms or discharge letters.
Observation / containment	Explain observation level, rationale, and duration constraints in simple language, grounded in the flow fields.	Observation/containment explanation for patient-facing communications and team handoffs.

This design mirrors the pattern already used for structured scales: numerical scores and severity bands are computed deterministically in TypeScript and Python, then included in AI contexts as read-only inputs rather than as quantities that the model is asked to infer.

7.4.3 Agitation episode narrative as an AI task

The agitation flow illustrates the pipeline concretely. Let $x_{\text{agit}} \in X_{\text{agit}}$ be a completed agitation form and t_{insert} its insertion time. The builder

$$\pi_{\text{agit}} = b_{\text{agit}}(x_{\text{agit}}, t_{\text{insert}})$$

produces a neutral paragraph that is inserted into the encounter note and stored in `completedRuns`. A simplified JSON record emitted by enc_{agit} might look as follows:

```
{
  "runId": "agitation-1716228000000",
  "flowId": "agitation",
  "insertedAt": "2025-05-20T15:00:00Z",
  "injuryRiskProfile": "verbal_only",
  "medicalContributorsConsidered": "delirium ruled out; pain considered",
  "deescalationTechniques": [
    "verbal de-escalation/calm tone",
    "calm environment/supportive setting"
  ],
  "escalationTypeDiscussed": "prn_med_offer",
  "postInterventionStatus": "calmer; no ongoing threat",
```

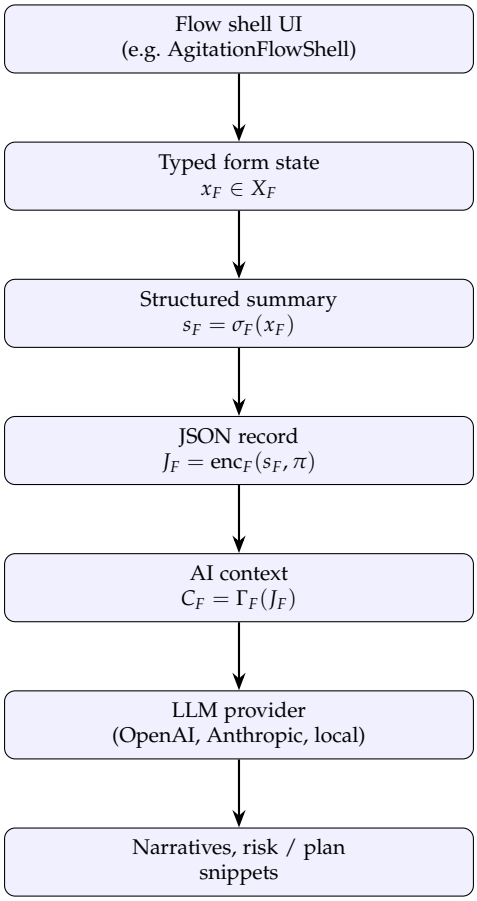


Figure 15: *Flow → AI pipeline. Completed flows are mapped from typed form state to structured summaries, then to JSON records and AI-ready contexts before being sent to the provider. The layout fits within the A4 text block without horizontal overflow.*

```
  "reassessmentPlan": "reassess within 30-60 minutes",
  "staffNotified": ["primary RN", "charge nurse"],
  "paragraph": "Agitation episode was documented this encounter..."
}
```

The agitation-specific context builder then constructs

$$C_{\text{agit}} = H_{\text{agit}} \,\|\, \text{serialize}\big(J_{\text{agit}}\big) \,\|\, P_{\text{MSE}},$$

where J_{agit} is the JSON object above, H_{agit} is a brief header describing the clinical scenario, and P_{MSE} is the MSE scaffold. The downstream task prompt is typically of the form:

> *Using only the structured facts and paragraph above, draft a concise, neutral narrative of the agitation episode (3–5 sentences) and 3–5 bullet points summarising current risk, protective factors, and the observation / medication plan. Do not introduce new facts.*

Because all clinically relevant content originates from J_{agit} and π_{agit}, the LLM cannot invent new risk elements or de-escalation steps; it only reorganises and rephrases the information already encoded by the flow. This preserves a clear division of labour: the flow framework determines *what* is true and clinically recorded, while the AI stack helps clinicians with *how* that information is communicated in notes, handoffs, or letters.

7.5 Formalisation of Flows

The flows subsystem can be formalised as a family of small, directed acyclic graphs (DAGs) that sit between the user interface and the encounter data model. Each flow F (e.g. safety, agitation, capacity) defines a finite set of states corresponding to form steps, together with the allowed transitions between them, and a deterministic mapping from a completed path to a typed data object. This graph-theoretic view makes it possible to reason about reachability, validation, and AI integration in a uniform way.

7.5.1 Graph-theoretic model of a flow

For a given flow F, we write

$$F = \big(V_F,\, E_F,\, v_F^{(0)},\, V_F^{\text{term}},\, \mathcal{D}_F,\, \lambda_F,\, \rho_F\big), \tag{7.4}$$

where:

- V_F is a finite set of nodes, each node $v \in V_F$ representing a *step* in the flow (e.g. "Baseline Behaviour", "De-escalation Attempts", ...). In code, these correspond to entries in the `STEPS` arrays in the shell components (e.g. `AgitationFlowShell.tsx`) and to the `StepPill` type in `flowTypes.ts`.
- $E_F \subseteq V_F \times V_F$ is a set of directed edges representing allowed transitions between steps. The graph (V_F, E_F) is acyclic when edges are oriented in the direction of semantic progression (clinical narrative). UI-level "back" actions traverse edges in the reverse direction but do not introduce cycles in this orientation.
- $v_F^{(0)} \in V_F$ is the unique initial node (start step) for F.
- $V_F^{\text{term}} \subseteq V_F$ is a non-empty set of terminal nodes at which the flow can be considered complete (e.g. an "Outcome" step).
- $\mathcal{D}_F = \{D_v : v \in V_F\}$ is a family of local data domains. For each step v, the field values captured by the corresponding TypeScript state (e.g. `AgitationFormState`) live in D_v.
- $\lambda_F : V_F \to \Sigma_F$ assigns each node a human-readable label (step title); in practice, this is the `key/label` pair used by `StepPills`.
- ρ_F is the accumulation map that takes a completed path together with local data and returns the flow-level summary object used in the registry and AI pipelines.

Since (V_F, E_F) is a DAG, the nodes can be topologically ordered so that all edges point from lower to higher indices. The current implementation takes the simple case where this order is given explicitly by the `STEPS` array; `goNext` and `goBack` in the shell components implement moves along this order, clamped to the valid range. Conditional branches (e.g. when certain steps are only relevant if a checkbox is set) can be modelled as subsets of E_F that are enabled or disabled by local guards without changing the underlying acyclic structure.

It is convenient to distinguish three abstract edge types:

- *Forward edges* $E_F^+ \subseteq E_F$, corresponding to "next" transitions.
- *Backward traversals*, which in the UI are implemented as moves to a previous index but are not included as forward-oriented edges in E_F.
- *Branch edges* $E_F^{\text{branch}} \subseteq E_F^+$, which allow the flow to skip or re-route around optional substeps (e.g. additional detail sections that may be omitted when not clinically relevant).

From the standpoint of formal analysis, only E_F^+ is required to define the DAG; backward movement is handled at the level of interaction logic (`StepPills` and shell state), and branches appear as alternative forward paths between nodes.

Table 30: *Key components of a flow F and their roles. Column widths are chosen to avoid page overflow.*

Symbol / analogue	Mathematical role	Implementation hook
V_F / `STEPS` array	Nodes (states / steps) of the flow DAG	Constant arrays in shell components (e.g. agitation, safety, capacity) defining keys and labels for each step.
E_F / implicit adjacency	Allowed step transitions	Implemented via constrained index changes (`goNext`, `goBack`) and conditional logic inside the shell.
D_v / form slice for step v	Local data domain for each node	Subset of the TypeScript form state type (e.g. `AgitationFormState`) that is semantically owned by step v.
λ_F / step labels	Human-readable state labels	`StepPill` labels rendered by `StepPills` with ARIA `role="tab"` semantics for accessibility.
ρ_F / outcome builder	Path-to-summary accumulator	Outcome builders (e.g. `buildAgitationOutcome`) and registry insertion actions (e.g. `appendFlowOutcomeAtSelection`).
V_F^{term} / "Outcome" step(s)	Terminal states; define when the flow is complete	Steps that enable finalisation actions (e.g. "Insert in note", "Mark run as completed") and produce a completed run entry.

7.5.2 Flow instances and data accumulation

An individual run of a flow F during an encounter corresponds to a path in the DAG together with local data objects collected at each node. Formally, a *flow instance* (or *run*) is a finite sequence

$$r = \big((v_0, d_0),\ (v_1, d_1),\ \ldots,\ (v_T, d_T)\big), \tag{7.5}$$

such that:

1. $v_0 = v_F^{(0)}$ is the initial node.
2. For each $k = 0, \ldots, T-1$ we have $(v_k, v_{k+1}) \in E_F^+$.
3. $d_k \in D_{v_k}$ is the local form data recorded at node v_k.
4. $v_T \in V_F^{\text{term}}$ when the run is finalised.

In the current UI, this path is realised as a combination of:

- the current step index stored in component state in the flow shell;
- moves between indices triggered by the `Next`, `Back`, or direct step pill selection controls; and
- the underlying React form state, which persists the d_k as the user moves between steps.

For each node $v \in V_F$, a local validation predicate

$$\text{valid}_v : D_v \longrightarrow \{0, 1\} \tag{7.6}$$

encodes whether the minimum documentation requirements for that step are met (e.g. at least one de-escalation technique selected, or a required text field completed). A run r is

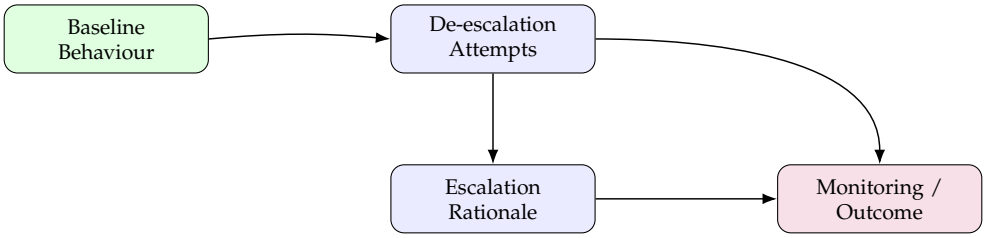


Figure 16: *Schematic of a simple flow DAG. Nodes correspond to steps in the agitation shell, and curved arrows represent allowed forward transitions, including a direct path from de-escalation attempts to monitoring and outcome. The layout and node sizes are constrained to remain within the text block.*

considered *complete* if

$$\text{complete}(r) = 1 \quad \Longleftrightarrow \quad v_T \in V_F^{\text{term}} \ \wedge \ \prod_{k=0}^{T} \text{valid}_{v_k}(d_k) = 1. \tag{7.7}$$

The user interface reflects this via step-level completion indicators (e.g. checkmarks on pills) and by enabling or disabling final actions based on validation state.

Once a run is complete, the accumulator ρ_F maps it into a flow-level summary object:

$$x_F = \rho_F(r) \in X_F, \tag{7.8}$$

where X_F is a typed TypeScript domain for flow summaries (e.g. the shape expected by the corresponding outcome builder). The builders in `Flows/builders/*.ts` then produce:

- a neutral narrative paragraph π_F for insertion into the note; and
- a structured record (run identifier, timestamps, flow identifier, and data fields) that is appended to the `completedRuns` array in the encounter state.

This is the same `completedRuns` structure used by the AI integration layer and export pipelines, ensuring that the graph-theoretic path r and its typed image x_F remain available for later inspection and reuse.

7.5.3 Flow graph schematic

Figure 16 illustrates a simplified agitation flow as a DAG, using orthogonal connections between steps. The example uses four states representing the shell steps in `AgitationFlowShell.tsx`. The edges encode the standard forward progression as well as an optional branch that allows the flow to move from a de-escalation step directly to the outcome when escalation detail is not required.

8 Session Timer and Clinical Audit Engine

8.1 Timer Engine

The session timer in SynapseCore is implemented as a pure TypeScript state machine in `src/centerpanel/components/`
`timerEngine.ts`. Its role is to maintain a precise notion of session time, laps and clinical segments, while remaining completely UI–agnostic. All rendering, keyboard shortcuts and visual effects in the timer modal are handled separately by `TimerModal.tsx`, which simply

invokes the pure transition functions (`start`, `pause`, `reset`, `tick`, `setMode`, `setZeroBehavior`, `startSegment`, `endCurrentSegment`, `addLap`) exposed by the engine.

In practice, the engine is driven either by a high–frequency loop based on `requestAnimationFrame` (for smooth on–screen updates) or by `setInterval` with a coarser step (for reduced–motion or background contexts). In both cases, the scheduler passes a measured time increment Δt in milliseconds into the pure `tick` function; all clinical logic and auditability depend only on the evolving timer state, not on the rendering strategy.

8.1.1 State model and invariants

The core type `TimerState` collects all information the engine needs to reason about a session:

$$\sigma_t = \big(m_t,\, p_t,\, t_t^{\mathrm{el}},\, t_t^{\mathrm{rem}},\, L_t,\, \mathcal{S}_t,\, z_t,\, a_t\big), \tag{8.1}$$

where, at discrete logical time step $t \in \mathbb{N}$:

- $m_t \in \{\text{stopwatch}, \text{countdown}\}$ is the timer *mode* (`mode`);
- $p_t \in \{\text{idle}, \text{running}, \text{paused}, \text{finished}\}$ is the current *phase* (`phase`);
- $t_t^{\mathrm{el}} \in \mathbb{R}_{\geq 0}$ is the cumulative elapsed time in milliseconds (`elapsedMs`);
- $t_t^{\mathrm{rem}} \in \mathbb{R}_{\geq 0}$ is the remaining countdown time in milliseconds (`remainingMs`), with an initial value $t^{\mathrm{init}} \geq 0$ (`countdownInitialMs`);
- L_t is the ordered list of laps (`laps`), each lap $\ell = (\mathrm{id}, t_\ell, \mathrm{label})$ storing an identifier, a timestamp t_ℓ and an optional label;
- $\mathcal{S}_t$ is the list of clinical segments (`segments`), each segment $s = (\mathrm{id}, k, t_{\mathrm{start}}, t_{\mathrm{end}})$ with kind $k \in \{\text{assessment}, \text{therapy}, \text{break}, \text{documentation}\}$ and optional end time;
- $z_t \in \{\text{finish}, \text{pause}, \text{keep}\}$ encodes the behaviour at zero (`zeroBehavior`);
- $a_t \in \{\text{true}, \text{false}\}$ (field `autoStopAtZero`) records whether the engine should automatically move to `"finished"` when $t_t^{\mathrm{rem}} = 0$.

Internally, the engine also maintains a monotonically increasing `_idCounter` used to generate stable identifiers for new laps and segments. All time quantities are clamped to non–negative values by a helper `clampNonNeg`, ensuring invariants such as $t_t^{\mathrm{el}} \geq 0$, $t_t^{\mathrm{rem}} \geq 0$ and segment durations $\max(0, t_{\mathrm{end}} - t_{\mathrm{start}})$ at every step.

To unify treatment of modes, the engine defines a derived "progress clock" $\pi_t = \texttt{progressMs}(\sigma_t)$ given by

$$\pi_t = \begin{cases} t_t^{\mathrm{el}}, & \text{if } m_t = \text{stopwatch}, \\ t^{\mathrm{init}} - t_t^{\mathrm{rem}}, & \text{if } m_t = \text{countdown}, \end{cases} \tag{8.2}$$

which is used for laps and segment boundaries regardless of the underlying mode.

8.1.2 Transition functions and piecewise dynamics

The engine exposes a small, composable set of pure transition functions:

- `start`: moves σ_t from `"idle"` or `"paused"` into `"running"`, honouring the current `mode` and `zeroBehavior`;
- `pause`: transitions from `"running"` to `"paused"` without altering any time fields;
- `reset`: returns the session to `"idle"`, zeroing `elapsedMs`, restoring `remainingMs` to `countdownInitialMs`, and clearing laps and segments;
- `setMode`: switches between `"stopwatch"` and `"countdown"`, optionally updating the initial countdown budget;

Table 31: *Key fields of* `TimerState` *and their roles in the session timer engine.*

Field	Type	Role
`mode`	`"stopwatch"` or `"countdown"`	Selects whether t_t^{el} grows unbounded or the engine tracks time against a fixed initial budget t^{init}.
`phase`	`"idle"` `"running"` `"paused"` `"finished"`	Encodes the control state; transition functions never modify time if `phase` is not `"running"`.
`elapsedMs`	`number`	Accumulates elapsed time for stopwatch mode; in countdown mode it complements `remainingMs` to capture total session duration.
`countdownInitialMs`	`number`	Stores the initial countdown budget t^{init}; used both to initialise and to reset `remainingMs`.
`remainingMs`	`number`	Remaining time in countdown mode; always kept ≥ 0 by clamping and drives zero–behaviour transitions.
`laps`	`Lap[]`	Time–stamped intra–session events (e.g. medication, risk discussion, change of therapist) forming the basis of an audit trail.
`segments`	`Segment[]`	Encodes labelled blocks such as *assessment* vs. *therapy*; used later to compute duration statistics per segment kind.
`zeroBehavior`	`"finish"` `"pause"` `"keep"`	Determines how the engine responds when `remainingMs` hits zero: stop, pause, or continue into negative time (for explicit overrun).
`autoStopAtZero`	`boolean`	Convenience flag mirroring `zeroBehavior` for UI logic; when `true`, the countdown session is considered complete at zero.

- `setZeroBehavior`: adjusts how the engine behaves when the countdown reaches zero;
- `tick`: advances time by a supplied increment Δt;
- `addLap`, `startSegment`, `endCurrentSegment`: mutate the lap and segment lists using the unified clock π_t.

The temporal evolution is entirely captured by `tick`. For a given state σ_t and a non–negative increment Δt, the next state σ_{t+1} is defined by:

$$\begin{aligned} &\sigma_{t+1} = \sigma_t, \\ &\sigma_{t+1} = \sigma_t \text{ with } t_{t+1}^{\text{el}} = t_t^{\text{el}} + \Delta t, \\ &\sigma_{t+1} = \sigma_t \text{ with } t_{t+1}^{\text{rem}} = \max\{0,\, t_t^{\text{rem}} - \Delta t\} \text{ and phase updated according to } z_t. \end{aligned} \tag{8.3}$$

The three lines correspond respectively to: (i) the case $\Delta t \leq 0$ or $p_t \neq$ running; (ii) $m_t =$ stopwatch; and (iii) $m_t =$ countdown.

When $m_t =$ countdown and $t_{t+1}^{\text{rem}} = 0$, the engine inspects `zeroBehavior`:

$$p_{t+1} = \begin{cases} \text{finished}, & z_t = \text{finish}, \\ \text{paused}, & z_t = \text{pause}, \\ \text{running}, & z_t = \text{keep}. \end{cases} \tag{8.4}$$

This design yields a piecewise linear trajectory $t \mapsto t_t^{\text{el}}$ (or $t \mapsto t_t^{\text{rem}}$) over discrete time steps. In continuous time $s \in \mathbb{R}_{\geq 0}$, driven by a scheduler that generates a sequence of increments Δt_i such that $\sum_i \Delta t_i = s$, the elapsed time is simply $t^{\text{el}}(s) = \sum_i \Delta t_i$ in stopwatch mode and the remaining countdown time is $t^{\text{rem}}(s) = \max\{0, t^{\text{init}} - \sum_i \Delta t_i\}$.

8.1.3 Selectors, snapshots and clinical aggregates

For integration with the rest of SynapseCore, the engine provides a set of *selectors*—pure functions that derive clinically relevant quantities from σ_t without mutating it. Examples include:

- `progressMs` and `remainingMsSelector`, which expose π_t and t_t^{rem} for rendering and alarms;
- `segmentTotals`, which aggregates total milliseconds spent in assessment, therapy, break and documentation segments, later used by the Clinical Snapshot Strip and export recipes;
- `fmtHMS`, which converts milliseconds into a fixed–width clock string `HH:MM:SS` or `MM:SS` suitable for the hero readout.

Persistence is handled via a small `TimerSnapshot` schema (`v: 2` plus a subset of fields from `TimerState`), together with `snapshotFromState` and `stateFromSnapshot` helpers. This allows session timers to be suspended and resumed across page reloads or device sleep, while preserving laps, segments and zero–behaviour. Snapshotting is explicitly versioned so that future extensions (e.g. additional segment kinds, per–segment notes, or ML features) can be introduced without corrupting existing data.

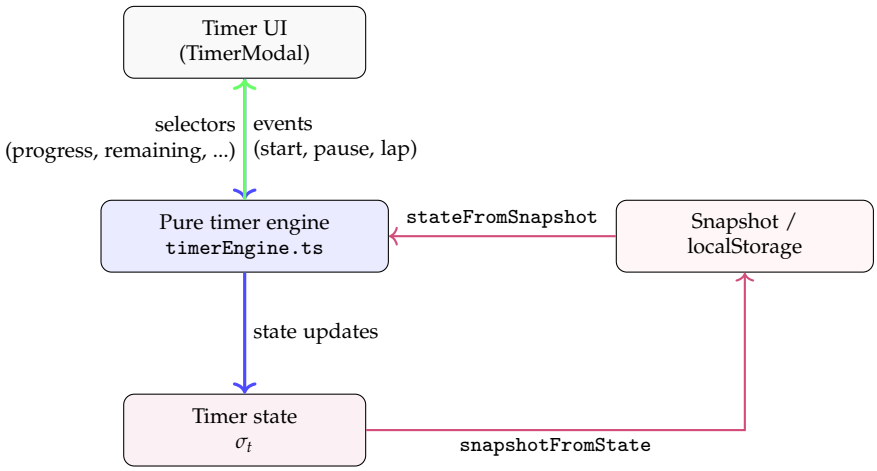


Figure 17: *High–level structure of the timer engine. The UI sends control events to the pure timer engine and consumes selector outputs, while snapshots mediate persistence across sessions.*

8.2 Timer Modal UI

The interactive timer modal is implemented in `centerpanel/components/TimerModal.tsx` and styled via `centerpanel/styles/timer.module.css`. Together with the shared timer engine (Section 8.1), this component provides a clinically oriented timekeeping surface that combines a high-contrast hero readout, a secondary countdown line, a clinical segment selector, and preset controls suitable for structured psychiatric encounters (assessment, therapy, breaks, documentation).

At any instant, the timer engine exposes a state $\sigma_t \in \mathcal{S}$ (cf. Section 8.1). The modal implements a pure rendering map

$$U : \mathcal{S} \to \mathcal{U}, \qquad u_t = U(\sigma_t) = \big(d_t, m_t, s_t, p_t, z_t, \ell_t, a_t\big),$$

where d_t denotes the hero time readout, m_t the active mode (stopwatch or countdown), s_t the current clinical segment, p_t the armed preset, z_t the zero-behaviour policy (`finish`, `pause`,

keep), ℓ_t the laps timeline, and a_t auxiliary UI flags (muted state, theming, audit indicators). The goal of the UI is to expose these degrees of freedom with minimal friction, preserving Synapse visual language and accessibility guarantees.

8.2.1 Layout and visual hierarchy

The modal renders as a glassmorphism card (`.sheet`) mounted in a dedicated `createPortal` container with a full-screen `.overlay`. The card width is constrained to $\min(0.96\,\texttt{vw}, 1280\text{px})$ and height to $0.92\,\texttt{vh}$, so that the timer remains readable but never exceeds the page or viewport bounds. The internal layout is a two-row grid:

- *Header row*: compact toolbar with title, patient and encounter context (via the registry state), theme badges, and the mode selector (Stopwatch versus Countdown).
- *Content row*: flexible grid containing (i) the hero readout and dual progress rings, (ii) the secondary countdown line and segment selector, (iii) the preset row and zero-behaviour controls, (iv) an optional audit and export panel on the right in wider viewports.

The hero readout uses a dedicated `.readoutGrid` style with tabular monospaced digits (Coder or a JetBrains Mono compatible stack) and responsive sizing

$$\text{font-size} = \text{clamp}\big(148\text{px}, f(W), 212\text{px}\big), \qquad f(W) = \frac{\min(W,\,1200) - 64}{9},$$

where W denotes the effective viewport width in pixels. This guarantees that the pattern `hh:mm:ss` always fits within the hero canvas, even on narrow displays or when embedded screenshots are scaled in the paper.

Under the hero digits, a secondary line renders a compact textual summary, for example "Countdown, 25 minutes armed, auto-pause at zero" or "Stopwatch, 18 minutes elapsed", tying the numeric display back to the underlying triple (m_t, p_t, z_t) without requiring the clinician to mentally reconstruct the configuration.

8.2.2 Stopwatch versus countdown modes

Mode selection is handled by an engine type $\texttt{Mode} \in \{\texttt{stopwatch}, \texttt{countdown}\}$ and the corresponding UI affordances:

- **Stopwatch mode**:
  The hero readout displays $d_t = \text{fmt}(\text{elapsedMs}(\sigma_t))$ for elapsed time. The secondary line highlights the active clinical segment and lap count (for example, "Therapy segment, 3 laps"). Presets are visually de-emphasised; keyboard shortcuts act on laps rather than presets.
- **Countdown mode**:
  The engine keeps a fixed $T_0 = t_0^{\text{countdown}}$ and a remaining time t_t^{rem}. The hero readout renders $d_t = \text{fmt}(\text{remainingMs}(\sigma_t))$, while the secondary line encodes the active preset p_t and zero-behaviour z_t. Mode switches while the timer is running are blocked with an accessible announcement ("Pause to switch mode.") to avoid accidental reconfiguration during a live session.

Formally, the mode-aware readout map is

$$d_t = \begin{cases} \text{fmt}\big(\text{elapsedMs}(\sigma_t)\big), & \text{if } m_t = \texttt{stopwatch}, \\ \text{fmt}\big(\text{remainingMs}(\sigma_t)\big), & \text{if } m_t = \texttt{countdown}, \end{cases}$$

where fmt converts milliseconds to an (hh, mm, ss) triple with optional hour suppression, and the functions elapsedMs and remainingMs are inherited from the engine (Section 8.1).

8.2.3 Clinical segment selector and presets

The clinical segmentation of a session is exposed via a chip-based radiogroup:

- Segments correspond to `SegmentKind` $\in \{$`assessment`, `therapy`, `break`, `documentation`$\}$.
- In the UI, these are rendered as a horizontally aligned group with `role="radiogroup"` and per-chip `role="radio"`, using `styles.segmentChips` and `styles.segActionBtn`.
- Clicking a chip invokes `Engine.startSegment` and publishes a `segment_change` event on the timer bus; a screen-reader update announces the new segment.

Countdown presets provide one-click access to commonly used durations (reflecting typical outpatient session blocks):

- Fixed presets at 10, 25, 45, and 50 minutes, plus a custom editor.
- Each preset sets $T_0 = m \times 60{,}000$ milliseconds and updates p_t, while enforcing guards such as "Pause to change presets." when the countdown is already running.
- Keyboard shortcuts $\{1, 2, 3, 4, 5\}$ mirror the visual presets, mapping to 10, 25, 45, 50 minutes or opening the custom dialogue.

From the engine viewpoint, arming a preset realises the update

$$\sigma_t \mapsto \sigma_t' = \text{setMode}\big(\sigma_t,\ \texttt{countdown},\ m \times 60{,}000\big),$$

followed by a persistence and audit step (Section 8.5). The UI ensures that this transition is always paired with a visual confirmation and a short textual toast.

8.2.4 Typography, theming, and button system

The timer CSS module defines a local token space $\{$`-tm-bg`, `-tm-fg`, `-tm-surface-1`, `-tm-hairline`, `-tm-btn-h`, $\ldots\}$ that is bridged to the global Synapse theme. Key decisions are:

- **Monospaced hero digits**: the hero readout uses a monospaced stack with `font-variant-numeric: tabular-nums` to guarantee that digit columns do not jitter when time updates.
- **Segmented controls**: mode and segment selectors share a segmented control style (`.seg`, `.segActive`) with hairline borders, soft accent overlays, and WCAG-compliant contrast in both light and dark skins.
- **Button system**: primary, secondary, and icon buttons in the modal reuse the Synapse button system (`.heroBtnPrimary`, `.ctrlBtn`, `.iconBtn`), ensuring consistent hit-target sizes, keyboard focus rings, and pressed states across the app.

The time-of-day theme hook `useTimeOfDayTheme` adjusts subtle background and glow accents (morning, afternoon, evening) without changing the core layout, preserving recognisability while giving the clinician a sense of temporal context across the clinical day.

Table 32: *Timer modes and primary UI affordances in the Synapse timer modal. Columns are left-aligned to improve readability and prevent visual overflow.*

Dimension	Stopwatch mode	Countdown mode
Hero readout	Elapsed time $d_t = \text{fmt}(\text{elapsedMs}(\sigma_t))$. Hours can be forced on/off via a persisted preference.	Remaining time $d_t = \text{fmt}(\text{remainingMs}(\sigma_t))$ until the armed duration T_0 is exhausted.
Secondary line	Emphasises active segment and number of laps; communicates whether the session is idle, running, paused, or finished.	Summarises armed preset (e.g. "25 min") and zero-behaviour (`pause`, `finish`, `keep`) in natural language.
Segment selector	Available in both modes; radiogroup over {*Assessment*, *Therapy*, *Break*, *Documentation*} with accessible keyboard handling.	Same segments; often used to structure a fixed 45 or 50 minute block into assessment, therapy, and documentation slices.
Presets	Visually present but de-emphasised; interaction focuses on start, pause, and laps.	Primary interaction surface for arming 10, 25, 45, or 50 minute blocks or entering a custom duration; guarded while the countdown is running.
Behaviour at zero	Not applicable; the stopwatch can run indefinitely until manually stopped.	Controlled by $z_t \in \{\texttt{finish}, \texttt{pause}, \texttt{keep}\}$, with matching auditory and visual cues when $t_t^{\text{rem}} = 0$.

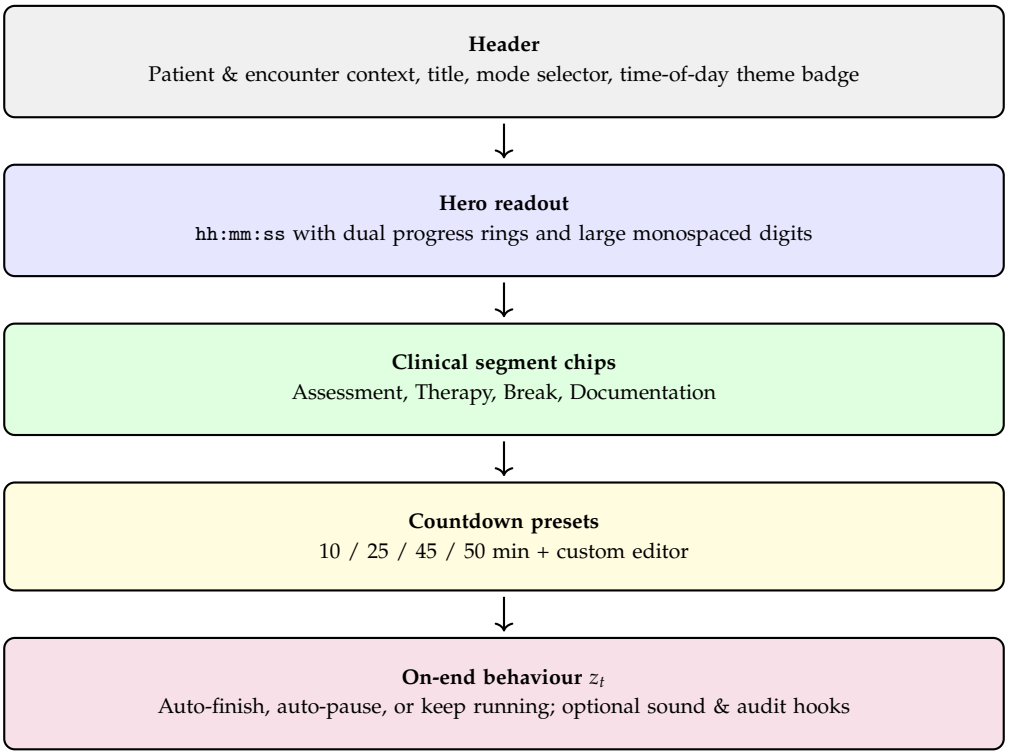


Figure 18: *Vertical schematic layout of the timer modal with enhanced visual clarity. Each functional block is represented as a distinct card with a soft colour grouping, forming a structured and clinically meaningful hierarchy that fits neatly within page margins.*

8.3 Laps and Clinical Events

8.3.1 Overview and motivation

In the timer subsystem, laps are treated as a lightweight but semantically rich representation of key clinical events during a session. Rather than being generic "splits" on a stopwatch, each lap is anchored to the internal `TimerState`, associated with a clinical segment, and exported as part of a canonical audit payload. This allows the same discrete markers to serve three purposes at once: real–time feedback for the clinician, retrospective documentation in notes, and structured features for downstream analytics.

At engine level, the TypeScript model in `timerEngine.ts` defines a lap as a timestamped marker on the continuous session timeline:

```
export interface Lap { id: string; atMs: number; label?: string; }
```

and the pure helper `addLap(s: TimerState, label?: string): TimerState` appends a new lap at the current progress time, computed by `progressMs(s)`. Because the timer engine is pure and side–effect free, laps remain consistent with the underlying `TimerState` regardless of whether the timer is in stopwatch or countdown mode, and regardless of how often the UI re-renders.

8.3.2 Engine–level representation

Formally, consider a session of total duration $T \geq 0$ and let

$$0 \leq t_1 < t_2 < \cdots < t_n \leq T$$

denote the strictly increasing sequence of lap times in milliseconds, as stored in the engine via the `atMs` field. For each lap we define a quadruple

$$L_i = (i, t_i, \Delta t_i, s_i),$$

where $i \in \{1, \ldots, n\}$ is the 1-based index, t_i is the absolute lap time since the start of the session, Δt_i is the split interval since the previous lap, and s_i is the segment label active at t_i.

The split is derived deterministically in the engine as

$$\Delta t_i = \begin{cases} t_1, & i = 1, \\ t_i - t_{i-1}, & i > 1, \end{cases} \tag{8.5}$$

which matches the runtime computation in both the mini–timeline component and the audit export code. In TypeScript, this is implemented by threading a mutable accumulator `prev` through the mapping over `t.laps`, with `split` set to `Math.max(0, l.atMs - prev)` and then updating `prev = l.atMs`.

The segment label s_i is inferred on demand from the current list of segments

$$\mathcal{S} = \{(k_j, a_j, b_j)\}_{j=1}^{m},$$

where $k_j \in \{\texttt{assessment}, \texttt{therapy}, \texttt{break}, \texttt{documentation}\}$ is a segment kind and $[a_j, b_j)$ is its interval in milliseconds, as stored in `TimerState.segments`. In the React implementation

(TimerModal.tsx), each lap time t_i is mapped to the unique segment whose interval contains it, if any:

$$s_i = \begin{cases} k_j & \text{if } \exists j \text{ with } a_j \le t_i < b_j, \\ \text{None} & \text{otherwise,} \end{cases} \tag{8.6}$$

with a UI fallback to the user–supplied textual label or the generic caption "Lap" when no enclosing segment is found. This gives each lap a coarse clinical context (for example assessment, therapy, break, documentation) without forcing the clinician to annotate segments manually.

For many analytic tasks it is convenient to view laps as a sequence of typed events

$$E_i = (t_i, c_i, s_i),$$

where c_i is an event code (for example *assessment_start*, *first_dose*, *de_escalation*) derived from the free–text label. The mapping from free–text labels into a small controlled vocabulary of event codes can be refined over time without changing the raw engine representation.

8.3.3 Canonical payload and audit export

For auditability and downstream analytics, laps are serialised into a canonical JSON payload alongside segments and session metadata. Inside TimerModal.tsx, the helper buildCanonicalPayload constructs an object of the form:

```
type CanonicalPayload = {
  engine: {
    mode: Engine.Mode;
    end_behavior: 'auto-pause' | 'stop' | 'keep';
  };
  meta: { app: 'therapy-timer'; schema_version: 1 };
  session: {
    started_at: string;
    ended_at: string | null;
    duration_ms: number;
    end_state: Engine.Phase;
  };
  segments: Array<{
    name: string;
    total_ms: number;
    pct: number;
    splits: number;
    avg_split_ms: number;
  }>;
  spans: Array<{ segment: string; start_ms: number; end_ms: number }>;
  laps: Array<{ index: number; at_ms: number; split_ms: number; label: string }>;
};
```

Here the lap sequence is encoded as $\big((i, t_i, \Delta t_i, \ell_i)\big)_{i=1}^{n}$, where index stores i, at_ms stores t_i, split_ms stores Δt_i as in (8.5), and label stores a normalised lap label ℓ_i (empty string if none has been provided). The same function also computes per–segment totals via Engine.segmentTotals(t), an array of span records { segment, start_ms, end_ms }, and

Table 33: *Formal representation of laps and their canonical JSON fields. Index i, timestamp t_i, delta Δt_i and segment label s_i are sufficient to reconstruct the mini–timeline and to drive audit exports.*

Component	Mathematical symbol	Canonical field(s)	Semantics
Index	$i \in \{1, \dots, n\}$	`index`	Stable 1-based position of the lap within the session.
Timestamp	$t_i \in \mathbb{R}_{\geq 0}$	`at_ms`	Milliseconds since session start at which the lap was recorded.
Delta / split	Δt_i (Eq. 8.5)	`split_ms`	Interval since the previous lap; Δt_1 equals t_1.
Segment label	s_i (Eq. 8.6)	derived from `segments`	Segment kind active at t_i (assessment, therapy, break, documentation) used for chips and summaries.

Table 34: *Illustrative taxonomy of lap–encoded clinical events. This classification is not enforced by the engine but provides a stable vocabulary for analytics and decision support.*

Event class	Example label	Typical trigger	Derived metric
Assessment milestones	"Assessment start"	Initial engagement or completion of core history/mental state exam.	Time from session start to assessment completion.
Risk / agitation events	"De-escalation achieved"	Observable change in agitation or risk level.	Time from first agitation note to de-escalation.
Medication interventions	"First dose given"	Administration of psychotropic medication.	Latency from consent to first dose; dosing intervals.
Administrative transitions	"Documentation phase"	Transition into documentation or handover.	Proportion of session time spent on documentation.

simple summary statistics such as `pct` (percentage of active time in each segment) and `avg_split_ms`.

The mapping from the mathematical representation to the canonical JSON fields is summarised in Table 33. The table also makes explicit that the segment label s_i is derived from the segment list $\mathcal{S}$ rather than being stored redundantly with each lap.

From a clinical perspective, it is useful to distinguish broad classes of events that laps can represent. Table 34 illustrates a simple taxonomy aligned with the canonical payload.

8.3.4 Mini–timeline layout and accessibility

At UI level, laps are rendered not as a dense table but as a vertically–stacked mini–timeline that compresses the most important fields into three stable columns:

- left: lap index and segment chip (for example "2 — Therapy");
- middle: absolute time from session start (for example `00:17:10`);
- right: delta since the previous lap (for example `+11:40`).

This layout is implemented in `TimerModal.tsx` via a flexbox row per lap inside a container with `role="list"` and `aria-label="Lap timeline"`, and styled by the `lapsTimeline` class in `timer.module.css`. The newest lap is given a transient highlight (a CSS "flash") so that clinicians can confirm that the event was captured. Because the timeline is strictly vertical and the content of each column is bounded, it remains readable even for long sessions with many laps and on narrow viewports.

The empty state is explicitly treated as part of the design. When `t.laps.length === 0`, the laps card shows an instructional message rather than a blank pane:

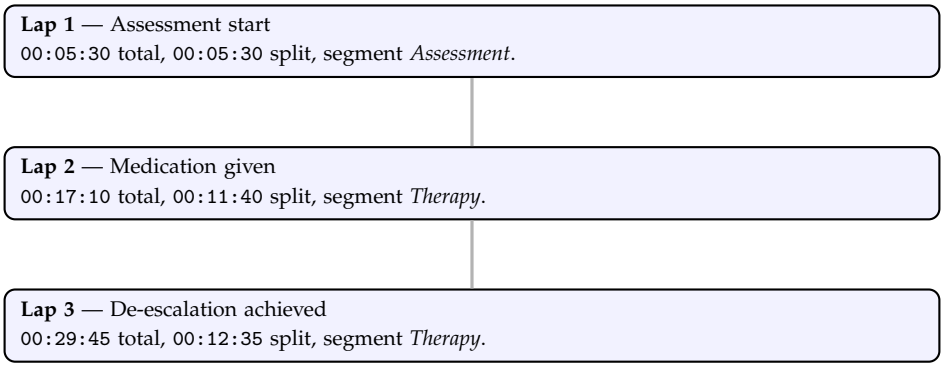


Figure 19: *Vertical mini–timeline of laps. Each row encodes the lap index and segment label on the left, total time since session start in the middle, and the delta since the previous lap on the right.*

No laps yet. Use the Lap shortcut (L) to record splits.

In practice, this copy is intended to nudge clinicians towards marking key clinical milestones (for example start of assessment, first medication dose, de-escalation, change in risk level) rather than pressing Lap at arbitrary times.

Figure 19 illustrates a typical vertical mini–timeline constructed from three such events.

8.3.5 Derived metrics and future extensions

Because the canonical payload exposes the full lap sequence and segment–level totals, it is straightforward to compute higher–order metrics from a set of sessions. For example, given a collection of sessions $\{s^{(k)}\}$ with per–session lap sequences $\{L_i^{(k)}\}$, one can define:

- median time to first medication in an agitation episode;
- distribution of documentation time as a fraction of total time;
- average number of laps per segment kind, stratified by diagnosis or setting.

In future work, these metrics could be used to drive feedback dashboards for trainees (for example showing trends in documentation efficiency) or to provide context–aware prompts in the AI assistant (for example suggesting a safety checklist when a “de-escalation” lap is recorded). In this way, laps serve as a bridge between the moment–to–moment rhythm of a live session and the aggregate, quality–improvement view that a digital psychiatry workbench aims to provide.

8.4 Timer Hooks and ML Integration

The core timer engine in `centerpanel/components/timerEngine.ts` is intentionally kept as a small, deterministic transition system. Around this nucleus, SynapseCore layers a collection of hooks in `centerpanel/timerHooks/` that (i) expose a typed event bus for intra-modal communication, (ii) attach auxiliary behaviours such as queueing, calendar alignment, persistence and metronome control, and (iii) bridge the resulting session telemetry into a local TensorFlow.js model via `useSessionML`. The objective is to enrich the session with proactive, data-driven coaching while keeping the underlying timer state auditable, replayable, and free from side effects.

8.4.1 Typed timer event bus

The file `timerHooks/types.ts` defines a finite set of event types `TimerBusEventType` and a payload map `TimerBusPayloadMap` ranging over core timer actions (`'start'`, `'pause'`, `'lap'`, `'finish'`, `'segment_change'`, ...) as well as higher-level hook events (`'queue_add'`, `'calendar_attach'`, `'metronome_on'`, `'hook_error'`, etc.). Formally, let $\mathcal{E}$ denote the finite set of event tags and, for each $e \in \mathcal{E}$, let $\mathcal{P}_e$ be the corresponding payload space (e.g. $\mathcal{P}_{\text{lap}} = \{\text{segment label}, \Delta t\}$). The timer bus is then the pair of operations

$$\texttt{publish}: \ \mathcal{E} \times \mathcal{P} \longrightarrow \mathbf{1}, \tag{8.7}$$

$$\texttt{subscribe}: \ \mathcal{E} \times (\mathcal{P} \to \mathbf{1}) \longrightarrow \mathcal{U}, \tag{8.8}$$

where $\mathcal{P} = \bigsqcup_{e \in \mathcal{E}} \mathcal{P}_e$ is the disjoint union of payload spaces, and $\mathcal{U}$ is the set of unsubscribe functions.

The concrete implementation in `timerHooks/eventBus.ts` uses a `Map<TimerBusEventType, Set<Function>>` to store listeners. The `createTimerEventBus` constructor returns an object `TimerEventBus` whose `publish` method simply iterates over the subscribed listeners for a given event and attempts to invoke them (type-cast as `TimerBusListener<typeof type>`). Any listener-level errors are caught and ignored, preventing a single faulty hook from destabilising the timer modal.

The React hook `useTimerBus` in `timerHooks/useTimerBus.ts` ensures that each mounted `TimerModal` owns exactly one bus instance stored in a ref:

$$\texttt{bus}_{\text{modal}} \in \texttt{TimerEventBus},$$

and that this bus is reused across renders. Additional hooks can then subscribe to specific slices of the event stream via `useTimerBusSubscription(bus, type, fn)`, which registers a listener on mount and disposes it on unmount. Conceptually, the timer event bus defines a typed event log

$$\mathcal{L} = \left\{ (t_i, e_i, p_i) \right\}_{i=1}^{N},$$

where t_i is a wall-clock time, $e_i \in \mathcal{E}$ an event tag, and $p_i \in \mathcal{P}_{e_i}$ the payload. Hooks downstream of the bus implement pure(ish) functions over $\mathcal{L}$, such as computing queue state, metronome pulses, or training data for the session model.

8.4.2 Queue, calendar, and persistence hooks

The queue and calendar hooks demonstrate how timer events are lifted into higher-level clinical workflows while remaining strictly local to the browser.

Queue management and PII safeguards. The hook `useTimerQueue` in `timerHooks/queueHook.tsx` maintains a queue of documentation tasks (`QueueItem`) with fields such as `id`, `label`, `mode` (e.g. `'complete'`, `'keep'`, `'pause'`), `policy` (`'arm'` or `'auto'`) and timestamps. The queue state

$$Q_t = \left(\text{items}_t,\ \text{active\_queue\_id}_t\right)$$

is initialised from, and persisted to, localStorage via the small helper `makePersist<QueueState>('queue',1)` defined in `timerHooks/persist.ts`. The persist helper serialises a document $\text{doc} = (\text{schema\_version}, \text{data}, \text{updated\_at})$ to a namespaced key

$$k_{\text{queue}} = \texttt{timerModal.hook.queue.v1},$$

so that upgrading a hook can be handled by bumping the schema version and providing a new default.

Crucially, `useTimerQueue` performs a lightweight heuristic scan for suspicious personally identifiable information (PII) in `label` text (e.g. long digit sequences, e-mail patterns, name-like tokens). If the scan triggers, the hook can decline to persist the item and emit a `'hook_error'` event via the bus. This allows the queue to be used for labels such as "translate CBT homework" or "summarise intake note" while discouraging the storage of raw identifiers inside the timer payloads.

Calendar alignment. The calendar helper in `timerHooks/calendar.ts` links a running timer session to an external appointment object without importing the appointment's raw details. A `CalendarLink` consists of an `event_hash` (computed via `sha256Hex` over an external `UID` or iCal snippet), start and end timestamps, and an optional short code. The helper persists this link using `makePersist<CalendarLink | null>('calendar', 1)` and exposes operations to attach and detach calendar events. When a link is active, timer hooks can enrich audit exports with a stable reference to the originating calendar event, without embedding PHI-laden payloads.

Shared persistence layer. The generic persistence helper `makePersist<T>` centralises the logic of loading, saving, and versioning localStorage-backed state. For any namespaced key ns, the helper implements

$$\texttt{load}_{\text{ns}} : T \to T, \qquad \texttt{save}_{\text{ns}} : T \to \mathbf{1},$$

with defensive guards around JSON parsing and localStorage availability. This pattern is reused across queue, calendar, and other future timer hooks, ensuring that local-only session intelligence can accumulate over time without dependence on a server-side database.

Metronome driver. The class `Metronome` in `timerHooks/metronome.ts` implements a small Web Audio–based scheduler that drives "pulses" at a user-configurable tempo. Its state is given by

$$M_t = (\text{on}_t, \text{bpm}_t, \text{subdivision}_t, \text{volume}_t),$$

with $\text{subdivision}_t \in \{1, 2, 4\}$ controlling whether quarter, eighth or sixteenth notes are emitted. Internally, the metronome maintains a look-ahead loop that repeatedly:

1. Queries the current state via a `getStateFn`,
2. Schedules short audio clicks into an `AudioContext` up to a small horizon ($\approx 150\,\text{ms}$),
3. Emits `'metronome_tick'` events on the timer bus for visual overlays (e.g. animated rings in the timer hero).

Metronome control events (`'metronome_on'`, `'metronome_off'`, `'metronome_bpm_change'`, etc.) are treated like any other bus events, so higher-level hooks and UI components can orchestrate pacing feedback without tightly coupling to the audio implementation.

8.4.3 Session ML hook and TensorFlow.js model

Where the previous hooks operate purely at the level of events and local state, the React hook `useSessionML` in `timerHooks/useSessionML.ts` introduces a small, client-side predictive model that learns from historical timer usage. It exposes to the UI the tuple

(model, isTraining, trainingProgress, modelLoaded, historicalSessionCount, `predictNextSegment`, `getProact`

Table 35: *Summary of core timer hooks and their responsibilities in* `centerpanel/timerHooks/`.

Hook / module	Primary responsibility	Persistence / integration
`useTimerBus` / `eventBus.ts`	Typed, in-memory event bus for timer events and hook signals	Stateless in memory; shared across the timer modal instance
`useTimerQueue` / `queueHook.tsx`	Queue of pending documentation / export tasks with PII-aware labels	Persists `QueueState` via `makePersist('queue',1)`
`calendar.ts`	Attach/detach timer sessions to external calendar events	Persists `CalendarLink` via `makePersist('calendar',1)`; hashes external UIDs with `sha256Hex`
`persist.ts`	Namespaced, versioned localStorage helper for timer hooks	Encodes `PersistDoc<T>` at keys of the form `timerModal.hook.`*`ns`*`.v`*`n`*
`metronome.ts`	Web Audio–backed tempo pulses and `metronome_tick` events	Stateless beyond browser audio context; controlled via bus events
`useSessionML.ts`	TensorFlow.js session model and proactive suggestions	Reads and writes `consulton_session_history`; saves model to IndexedDB as `session-predictor`

which allows components such as `MLInsightsPanel` and `RealTimeCoaching` to surface training status and context-aware recommendations to the clinician.

The type `HistoricalSession` captures a completed session as

$$s = (\text{timestamp}, \text{segments}, \text{totalDuration}, \text{lapCount}, \text{pauseCount}),$$

where `segments` is an ordered list of labelled segments with durations in seconds. The helper `loadHistoricalSessions` reads an array of such objects from localStorage under the key `'consulton_session_history'`, and `saveSessionForTraining` appends a new element, truncating the list to the most recent 100 sessions. This provides a bounded, purely local training set.

Feature vector and prediction targets. From each historical session s, the function `extractFeatures(session, currentTime)` computes a fixed-length feature vector $\phi(s) \in \mathbb{R}^8$, summarised in Table 36. In symbolic form,

$$\phi(s) = \big(h,\, d,\, \bar{\tau},\, L,\, P,\, I_{\text{ther}},\, I_{\text{assess}},\, N\big), \tag{8.9}$$

where h and d encode time-of-day and day-of-week, $\bar{\tau}$ is the mean segment duration in hours, L and P are normalised lap and pause counts, I_{ther} and I_{assess} are binary indicators for therapy- and assessment-like content, and N captures the number of segments.

The prediction target combines (i) the type of the next likely segment and (ii) its expected duration. To this end, segment labels are mapped through `encodeSegmentName` to an integer $c \in \{0, \dots, 5\}$ representing one of $\{\text{Assessment}, \text{Therapy}, \text{Documentation}, \text{Break}, \text{Consultation}, \text{Other}\}$. The first segment of the subsequent session provides both a type code c_i and a duration δ_i (in hours), and

$$y_i = \big(c_i/5,\ \delta_i\big) \in [0,1] \times \mathbb{R}_+$$

serves as the label vector for the i-th training example.

Table 36: *Feature vector $\phi(\text{session})$ used by the timer session model in `useSessionML.ts`.*

Feature	Symbol	Description and normalisation
Time of day	h	Hour of the day divided by 24, $h = \text{hour}/24$.
Day of week	d	Day index (0–6) divided by 7, $d = \text{dow}/7$.
Mean segment duration	$\bar{\tau}$	$\bar{\tau} = (\text{totalDuration}/\max(1,n))/3600$, where n is the number of segments.
Lap count	L	$L = \text{lapCount}/10$.
Pause count	P	$P = \text{pauseCount}/5$.
Therapy indicator	I_{ther}	1 if any segment name contains "therapy", else 0.
Assessment indicator	I_{assess}	1 if any segment name contains "assessment", else 0.
Segment cardinality	N	$N = n/10$, with $n = \lvert\text{segments}\rvert$.

Model architecture and training objective. The hook constructs a small feedforward network

$$f_\theta : \mathbb{R}^8 \longrightarrow \mathbb{R}^2$$

using `@tensorflow/tfjs`, with an input layer of dimension 8, two hidden layers (16 and 8 units with `relu` activations and a 0.2 dropout between them), and a final linear layer of size 2. For a sequence of n historical sessions, the training set is obtained by pairing each session s_i with the next one s_{i+1}, so that $x_i = \phi(s_i)$ and y_i is derived from the first segment of s_{i+1} as above. The network is trained with the Adam optimiser and mean squared error loss:

$$\mathcal{L}(\theta) = \frac{1}{n-1} \sum_{i=1}^{n-1} \left\| f_\theta(x_i) - y_i \right\|_2^2, \tag{8.10}$$

using mini-batches and a small validation split. Training is only attempted once at least ten historical sessions are available, and progress is exposed via the state variable `trainingProgress` (in percent). After training, the model is persisted locally to IndexedDB under the URL `'indexeddb://session-predictor'` and reloaded on subsequent visits by `initModel`.

Inference and clinical surface. At inference time, the hook `predictNextSegment` accepts a `SessionPattern` describing the *current* session: time-of-day, day-of-week, cumulative duration, lap and pause counts, and the list of previous segment labels. It recomputes a feature vector in the same format as (8.9), feeds it through the loaded model, and decodes the output into a `SessionPrediction`

$$\hat{y} = \big(\widehat{\text{segment type}}, \widehat{\text{duration}}, \widehat{\text{confidence}}, \widehat{\text{reasoning}}, \widehat{\text{isAnomaly}}\big).$$

In the current implementation, confidence is derived heuristically and anomalies are flagged when the total session duration exceeds a fixed threshold (e.g. ≥ 4 hours), prompting suggestions such as "Consider taking a break." The function `getProactiveSuggestion` wraps this prediction into a short, clinician-facing string, while `generateReasoning` provides a human-readable explanation tailored to the time-of-day and predicted segment type (for example, recommending a 45-minute therapy block in the afternoon following a long assessment).

UI components like `MLInsightsPanel` and `RealTimeCoaching` consume these signals to display an "Active" versus "Data collection" status, training metrics, and inline coaching banners near the timer hero. Importantly, both the training data (`consulton_session_history`)

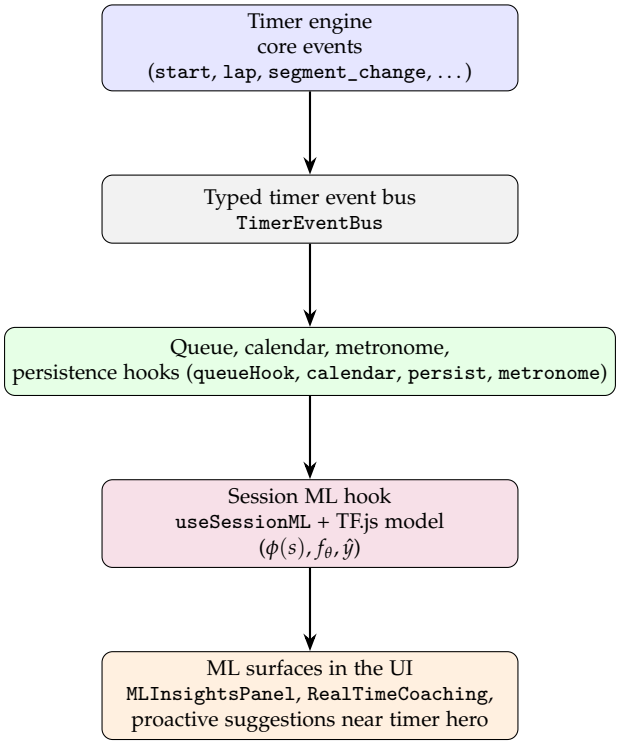


Figure 20: *Vertical pipeline for timer hooks and session-level ML. Core timer events are broadcast on a typed event bus, consumed by queue, calendar and metronome hooks, and summarised into feature vectors for a local TensorFlow.js model. The resulting predictions are surfaced in dedicated ML panels and real-time coaching overlays without leaving the clinician's browser.*

and the learned model (`session-predictor`) remain confined to the local browser environment; no session-level features or predictions are sent to remote servers, aligning the ML subsystem with the privacy constraints of digital psychiatry.

8.5 Audit and Export

The session timer, structured flows, and free-text notes generate a rich but heterogeneous stream of events during a consultation. From a clinical and forensic perspective, these must ultimately be condensed into a small number of well-structured artefacts: progress notes, risk summaries, consult letters, and—where necessary—formal audit trails that document what was done, when, and on the basis of which findings. In SynapseCore, this condensation is handled by an *audit and export* layer that sits at the boundary between real-time interaction components (timer, flows, notes, MBC engine) and the Tools/Consulton surface in the center panel (see Section 9). The goal is to ensure that every relevant event from the session can be traced, sampled, and rendered into structured Markdown or HTML with minimal manual friction, while remaining faithful to the underlying clinical logic.

Session audit record as a typed object

Formally, let $\mathcal{S}$ denote the set of sessions and, for each $s \in \mathcal{S}$, let $\mathcal{E}_s^{\text{timer}}$, $\mathcal{E}_s^{\text{flow}}$, $\mathcal{E}_s^{\text{note}}$, and $\mathcal{E}_s^{\text{mbc}}$ denote, respectively, the event streams emitted by the timer subsystem, structured flows, free-text notes, and measurement-based care engine during that session. The audit layer defines a session-level summary object $\text{Audit}(s)$ as:

$$\text{Audit}(s) \;=\; \big(h_s,\; \mathcal{T}_s,\; \mathcal{F}_s,\; \mathcal{N}_s,\; \mathcal{M}_s\big), \tag{8.11}$$

where:

- h_s is a header block containing de-identified session metadata (e.g. pseudo-identifier, date/time, clinician role, clinic/site);
- $\mathcal{T}_s$ is the timer-derived session timeline, including global start/end, segment boundaries, and labelled laps;
- $\mathcal{F}_s$ is the collection of flow-derived outcomes, containing the final state of each engaged flow (e.g. agitation, capacity, safety, catatonia, observation);
- $\mathcal{N}_s$ is the curated note corpus, including structured note fragments, AI-generated narratives explicitly accepted by the clinician, and any manually entered free text;
- $\mathcal{M}_s$ summarises MBC outputs (scale results, bands, flags) that were active or updated in the session.

At the TypeScript level, this corresponds to a composite type that exposes only immutable, read-only views of its components to the export machinery, ensuring that export recipes cannot accidentally mutate the underlying timer or flow state. The audit record thus acts as a single source of truth for *what happened in the session*, decoupled from any particular textual format (Markdown, HTML, or a future FHIR-compatible representation).

The relationship between raw events and the audit record can be written as a deterministic aggregation map

$$\Phi_{\text{audit}} : \left(\mathcal{E}_s^{\text{timer}},\ \mathcal{E}_s^{\text{flow}},\ \mathcal{E}_s^{\text{note}},\ \mathcal{E}_s^{\text{mbc}}\right) \longmapsto \text{Audit}(s), \tag{8.12}$$

implemented in practice by a small set of pure helper functions that normalise timestamps, flatten flow state into outcome objects, and attach MBC-derived risk and severity bands to the relevant segments of the session timeline.

Core schema and provenance

For clinical audit and medico-legal purposes it is essential that each field in the exported report be traceable back to its originating subsystem. In SynapseCore, this is handled by a stable schema for $\text{Audit}(s)$, which tags each field with its provenance and intended use. A simplified view of this schema is shown in Table 37.

By keeping the audit schema small and stable, export recipes in the Tools module can operate against a well-defined interface, even as individual flows or UI components evolve. For example, adding a new flow (e.g. for substance use or capacity-specific jurisdictional forms) simply extends $\mathcal{F}_s$ with an additional outcome type; existing export recipes continue to function as long as they either ignore unknown outcome types or provide generic fallbacks.

From audit object to Markdown and HTML exports

The final step is to transform $\text{Audit}(s)$ into user-facing artefacts. In SynapseCore, this is accomplished by a family of pure formatting functions

$$\Psi_r : \text{Audit}(s) \longrightarrow D_r(s), \tag{8.13}$$

where r indexes the export recipe (e.g. “progress-note”, “consult-letter”, “risk-summary”) and $D_r(s)$ denotes the document instance in a specific target format. In the current implementation these targets are primarily Markdown and HTML, produced by dedicated helpers in the Tools/Consulton subsystem that:

E1 construct a *scope object* that exposes $\text{Audit}(s)$ plus any additional contextual data (e.g. clinician preferences, institution name);

Field (within Audit(s))	Primary source subsystem	Example content / role in exports
Header h_s	Session metadata store	De-identified ID, date/time, location, clinician role; anchors the exported document and permits longitudinal linkage.
Timeline $\mathcal{T}_s$	Timer engine + laps	Ordered list of segments and laps with timestamps, duration summaries, and qualitative labels (e.g. "MSE", "Risk discussion").
Flow outcomes $\mathcal{F}_s$	Flow builders (agitation, capacity, safety, ...)	Normalised outcome objects capturing risk ratings, protective factors, agreed plans, and structured intervention summaries.
Notes $\mathcal{N}_s$	Notes editor + AI panel	Free-text narrative accepted by the clinician, with optional markers indicating which paragraphs originated from AI suggestions.
MBC summary $\mathcal{M}_s$	MBC engine	Scale names, total scores, severity bands, and flags; used to populate MBC sections in progress notes and letters.

Table 37: *Core components of the session audit record* Audit(s)*, including their primary subsystems and roles in downstream export recipes.*

E2 apply a recipe-specific formatter that renders the scope into a structured document tree (sections, headings, bullet lists, tables);

E3 serialise the document tree into Markdown or HTML, ensuring that sensitive identifiers are omitted or pseudonymised according to the current de-identification settings;

E4 push the resulting artefact into the Tools export inbox, where the clinician can preview, edit, copy to clipboard, or save to file.

Because both Φ_{audit} and Ψ_r are implemented as side-effect-free transformations, the same session can be re-exported under different recipes without risk of divergence in the underlying data.

Vertical data flow into the Tools/Consulton surface

Figure 21 summarises this audit and export path as a vertical data flow from real-time subsystems at the top to the Tools and Consulton surfaces at the bottom. The emphasis is on aligning the conceptual layers with the actual composition of React/TypeScript components: each box represents a small, testable module rather than a monolithic export engine.

Clinically, this design allows the psychiatrist to treat the audit and export machinery as a transparent layer: the timer, flows, and notes are used in a natural manner during the session, and at the end the Tools/Consulton surface offers a menu of export options that are guaranteed to be consistent with the underlying session history. From an engineering perspective, the fact that all exports factor through a typed audit object greatly simplifies testing, observability, and future extensions such as FHIR-compatible exports or institution-specific documentation templates.

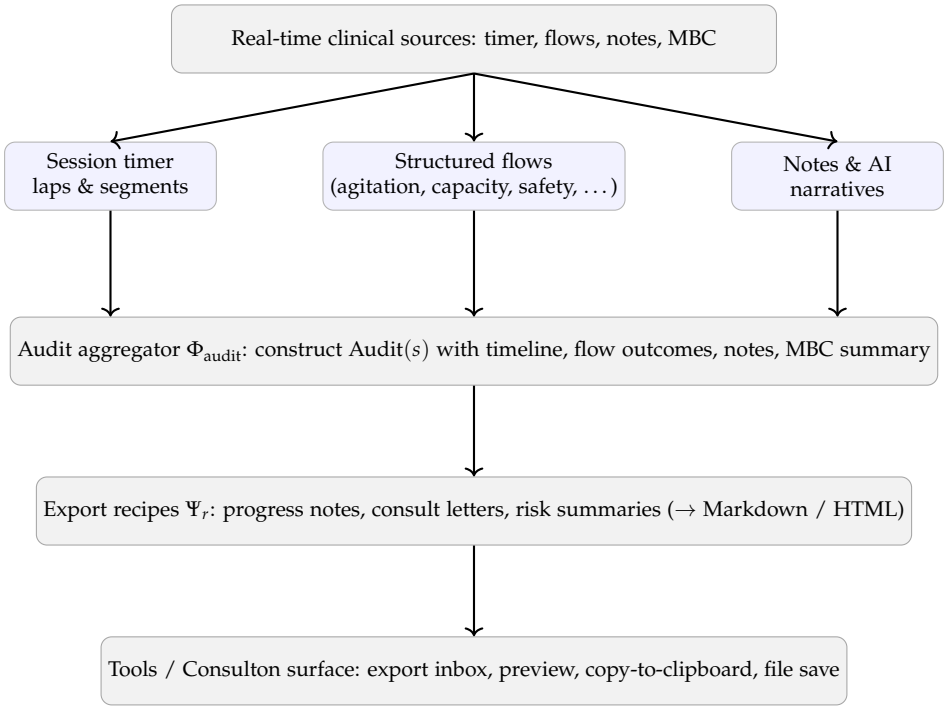


Figure 21: *Vertical audit and export pipeline in SynapseCore. Timer, flows, notes, and MBC outputs feed into an audit aggregator that constructs a session-level audit record* Audit(s)*. Export recipes then transform this into Markdown or HTML documents surfaced through the Tools/Consulton module.*

9 Clinical Tools and Consulton Module

9.1 Tools Framework

The Clinical Tools and Consulton module is the final assembly point where SynapseCore turns raw, intra-session activity into concrete artefacts that can enter the medical record. While the timer, flows, and MBC layers operate *within* the temporal dynamics of the consultation, the Tools framework operates at the level of *sessions as documents*. It exposes a small, stable set of user interactions for selecting a patient and session, invoking export recipes, and managing a queue of generated documents.

From the code perspective, the framework is implemented under `centerpanel/Tools/` and is deliberately small: `ToolsPatientList.tsx` handles patient and session scoping, `ToolsActionPanel.tsx` acts as the primary toolbar, `ExportBar.tsx` renders the export queue, and `exportInbox.ts` defines the underlying export store. Together, these components provide a clinically intuitive view over a rigorously typed export pipeline that is grounded in the audit object Audit(s) described in Section 8.5.

9.1.1 Scope: patient and session selection

A central design requirement for the Tools view is that every exportable document must be anchored to a uniquely identified clinical encounter. Let $\mathcal{P}$ denote the set of patients and $\mathcal{S}(p)$ the set of sessions for patient $p \in \mathcal{P}$. The Tools framework always maintains a *scope pair*

$$(p^*, s^*) \quad \text{with} \quad p^* \in \mathcal{P},\ s^* \in \mathcal{S}(p^*), \tag{9.1}$$

representing the patient and session currently selected in `ToolsPatientList.tsx`. This scope pair is the only entry point into the audit engine:

$$(p^*, s^*) \longmapsto \text{Audit}(s^*), \tag{9.2}$$

so that all downstream actions (export recipes, AI-assisted drafting, and clipboard operations) are guaranteed to reference a single, well-defined consultation.

At the UI level, `ToolsPatientList.tsx` renders a hierarchical list: patients form the primary rows, and expanding a patient reveals associated sessions (for example, by date and high-level label such as "initial assessment", "follow-up", or "crisis review"). Clicking on a session row triggers two operations:

S1 update the global Tools scope store with the new pair (p^*, s^*);
S2 request (or recompute) the audit object $\text{Audit}(s^*)$ from the shared audit engine.

The resulting React state change causes `ToolsActionPanel` and `ExportBar` to re-render in a way that is fully consistent with the new scope: previously generated exports remain visible but are visually separated from exports belonging to other sessions (for example by patient label and session date).

Clinically, this explicit scoping protects against a subtle but important failure mode: copying a progress note into the wrong chart. By forcing every export to flow through a consciously selected (p^*, s^*), the framework makes it difficult to accidentally mix content from two different patients or from two temporally distant consultations.

9.1.2 Action toolbar and export operations

Given a scoped session s^* and its associated audit object $\text{Audit}(s^*)$, `ToolsActionPanel.tsx` presents a set of export recipes $\mathcal{R} = \{r_1, \dots, r_k\}$. Each recipe corresponds to a clinically meaningful document type (e.g. progress note, consult letter, risk summary, discharge summary). At the type level, a recipe is a pure function

$$\Psi_r : \text{Audit}(s) \longrightarrow D_r(s), \tag{9.3}$$

where $D_r(s)$ is a structured document tree in an intermediate representation (paragraphs, headings, lists, tables) that can be serialised to Markdown or HTML.

The toolbar binds each visible button to a handler of the form

$$\texttt{onClick}_r = \lambda\big(\text{Audit}(s^*)\big).\ \mathsf{enqueueExport}\big(s^*, r, D_r(s^*)\big), \tag{9.4}$$

where `enqueueExport` is the mutator defined in `exportInbox.ts`. Concretely, clicking "Progress note" in the Tools view performs:

A1 retrieve the current $\text{Audit}(s^*)$ from the audit store;
A2 apply the selected recipe r_{note} to produce $D_{r_{\text{note}}}(s^*)$;
A3 construct a new export entry e bundling:
- identifiers (p^*, s^*),
- recipe identifier r_{note},
- the document payload $D_{r_{\text{note}}}(s^*)$,
- timestamps and initial status `pending`;

A4 push e into the export queue Q_t according to Equation (9.4).

Importantly, the toolbar is also aware of the runtime AI configuration (as described in earlier sections). If the chosen recipe delegates part of the drafting to the AI panel—for example,

constructing a narrative from structured flow outcomes—the transformation Ψ_r internally composes two steps: a deterministic normalisation of $\text{Audit}(s^*)$ into a prompt context, and an AI call via the provider-agnostic sampling mapper. The resulting text is tagged in the document tree so that downstream renderers can visually distinguish AI-suggested paragraphs from purely human-authored content.

9.1.3 Export inbox state and life-cycle

The `exportInbox.ts` module defines the shape and life-cycle of export entries. At time t, the export queue is an ordered list

$$Q_t = [\, e_1, e_2, \ldots, e_{n_t} \,], \tag{9.5}$$

with each entry e_i carrying:

- a unique `id`;
- patient and session identifiers (p_i, s_i);
- recipe identifier r_i;
- the document payload $D_{r_i}(s_i)$ in Markdown or HTML;
- a life-cycle status $\sigma_i \in \{\texttt{pending}, \texttt{edited}, \texttt{copied}, \texttt{discarded}\}$;
- timestamps for creation and last modification.

User actions in `ExportBar.tsx` induce transitions in the status field. A simplified state machine for a single entry is:

$$\texttt{pending} \xrightarrow{\text{edit}} \texttt{edited} \xrightarrow{\text{copy}} \texttt{copied} \xrightarrow{\text{discard}} \texttt{discarded}. \tag{9.6}$$

In practice, the implementation allows some shortcuts (for example, $\texttt{pending} \xrightarrow{\text{copy}} \texttt{copied}$) but enforces that `discarded` entries are no longer rendered, preventing accidental reuse.

From a clinical governance perspective, the export inbox provides a compact, on-screen history of what has been generated in the current Tools session. It is intentionally ephemeral: the queue is tied to the runtime of the application and does not attempt to be a permanent archive. This design choice keeps the boundary between *draft* and *record* clear: once the psychiatrist has copied a document into an EHR, the authoritative record lives in that external system, not in SynapseCore.

Table 38 summarises the main components participating in this life-cycle and the key invariants they maintain.

9.1.4 Vertical workflow within the Tools view

The Tools workspace is intentionally structured as a vertical workflow: selection at the top, generation in the middle, and delivery at the bottom. This mirrors the mental model of the psychiatrist: first decide *whose* session is being documented, then decide *what kind* of document to produce, and finally decide *where* that document should go. The successive layers of the workflow are illustrated in Figure 22.

By keeping the workflow narrow and vertically aligned, the Tools view implicitly encodes a set of best practices for documentation: exports are always scoped, always recipe-based, and always subject to explicit human review before leaving the application. This combination of clear clinical semantics and simple state transitions makes the Tools framework a reliable bridge between the dynamic SynapseCore workbench and the static but medico-legally binding world of the patient record.

Component / module	Location	Clinical and technical role
`ToolsPatientList`	`centerpanel/Tools/`	Renders patients and sessions; maintains the scope pair (p^*, s^*). Guarantees that every export is associated with a specific consultation.
`ToolsActionPanel`	`centerpanel/Tools/`	Presents export recipes and binds them to handlers that call Ψ_r on Audit(s^*) and enqueue the resulting documents. Respects runtime AI configuration when recipes are AI-assisted.
`ExportBar`	`centerpanel/Tools/`	Visualises the queue Q_t, exposes edit/copy/discard actions, and reflects status changes (`pending`, `edited`, `copied`, `discarded`) via iconography and muted colour changes.
`exportInbox`	`centerpanel/Tools/`	Implements the export queue store and pure reducers for enqueuing, updating, and pruning entries. Ensures that operations on Q_t are immutable and easily testable.

Table 38: *Key components of the Tools framework and their roles in managing the export life-cycle from scoped session selection to user-validated document artefacts.*

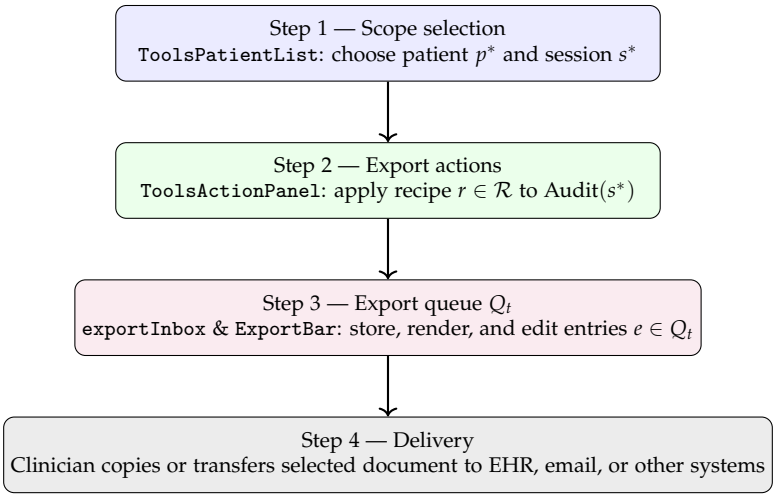


Figure 22: *Vertical workflow of the Tools framework. The clinician first selects a patient and session in `ToolsPatientList`, then uses `ToolsActionPanel` to generate documents via export recipes. These are materialised as entries in the export inbox and managed through `ExportBar` before being copied or transferred to external systems.*

9.2 ConsultonPanel and Sessions

The Consulton environment is the most narrative-oriented component of SynapseCore. Whereas the timer, flows, and Tools framework described in Section 9.1 operate primarily on structured data, the Consulton layer is designed for *clinical prose*: narrative consult letters, detailed case formulations, risk summaries, and other free-text documents that must nonetheless remain tightly coupled to the session audit record Audit(s).

In practice, this environment is implemented as `ConsultonPanel.tsx`, styled via `tools.module.css`, with longer-lived artefacts represented by the `ConsultSession` type in `ConsultonSessions.ts`. The panel provides three tightly integrated capabilities: (i) an ac-

cessible layout that juxtaposes clinician input with AI-generated output, (ii) a safe Markdown-to-HTML pipeline that sanitises all rendered content, and (iii) persistent *Consulton sessions* that track the evolution of a particular consult letter or report over time.

9.2.1 Panel layout and interaction design

From a user-interface perspective, `ConsultonPanel` is a grid-based layout created using the classes in `tools.module.css`. The root element `consultWrap` provides consistent padding and anchors the panel within the Center Panel shell. Immediately below the header (`consultHeader`), which shows the "Aegis Consult" brand, active model label, and policy badges (for example a "Redacted" pill when de-identification is enabled), the main body is organised into two primary columns defined by `consultBody`:

- a left column containing the input editor (`consultEditor`, `consultTextarea`) in which the psychiatrist writes instructions, pastes structured findings from flows, or drafts portions of the letter by hand; and
- a right column containing the output region, framed by `outHeader`, where AI-generated content is rendered as sanitised HTML and can be reviewed, selected, and copied.

On large screens, this layout is rendered as a two-column grid with a flexible left column and a constrained right column (`grid-template-columns: 1fr minmax(..., 360px)`). On smaller viewports (below a width threshold of approximately 1000 px), the CSS media queries defined in `tools.module.css` collapse the grid into a single column so that the input and output sections stack vertically. This ensures that the consult panel remains usable on laptops and tablets without requiring horizontal scrolling.

The header contains a small but clinically significant set of controls:

- A model label (for example "GPT-4o") that reflects the active AI backend, keeping the psychiatrist aware of which model family is being used for drafting.
- A "Documentation support" pill that explicitly frames the role of the AI as assisting with documentation rather than issuing directives.
- A "Redacted" pill when the effective de-identification policy is not `"none"`, signalling that the context derived from $\mathrm{Audit}(s)$ has undergone de-identification prior to being sent to the AI provider.
- A "Canary" control whose state is stored via `setConsultonCanaryOverride`; this toggles participation in controlled rollouts of new Consulton features.

Below the editor, `ConsultonPanel` renders the token budget meter (described in detail in Section 9.2.3) and a set of action buttons: "Generate", "Stop", "Copy", and optionally "Publish" to the Tools export inbox. These controls are wired to the AI streaming pipeline and to the `exportInbox` store such that any accepted draft can be materialised as an export entry and later combined with other documentation in the Tools framework.

9.2.2 Safe rendering of AI-generated content

Because Consulton routinely receives long, model-generated strings that may contain Markdown, HTML-like fragments, or user-supplied text, the rendering pipeline is designed to be aggressively defensive. The core transformation is implemented as:

$$\mathrm{HTML}_{\mathrm{safe}} = \mathsf{Sanitise}\big(\mathsf{MarkdownToHtml}(\mathrm{raw})\big), \tag{9.7}$$

where raw is the accumulated string buffer from the streaming AI response, MarkdownToHtml is implemented by the `marked` library, and Sanitise is provided by `DOMPurify`.

More concretely, when the streaming pipeline receives new chunks, the panel:

R1 concatenates them into an in-memory Markdown string raw_t;
R2 parses this string via MarkdownToHtml to obtain a preliminary HTML fragment;
R3 passes the fragment into `DOMPurify.sanitize` with a restricted set of allowed tags and attributes (for example `p`, `strong`, `em`, `ul`, `li`, `code`, `pre`, `table`, but not `script`, inline event handlers, or arbitrary attributes); and
R4 renders the resulting $\text{HTML}_{\text{safe}}$ into the output region using `dangerouslySetInnerHTML` inside a React component whose only responsibility is presentational.

The allowed-tag whitelist ensures that even if the upstream AI model were to produce potentially unsafe HTML (for example inline JavaScript, embedded iframes, or tracking pixels), these elements would be stripped before the content is injected into the DOM. Equation (9.7) is thus an explicit statement of the invariant that *all output must pass through a sanitiser before it can be displayed*.

Stylistically, the output pane uses classes such as `outHeader`, `outTitle`, and `outActions` to provide a clear visual frame around the generated content, with border radii and gradients aligned with the rest of the Center Panel design system. Status badges (for example streaming, success, warning, error) are implemented via `statusRow`, `statusStreaming`, `statusOk`, and related CSS classes, giving the psychiatrist immediate visual feedback about the state of the AI call and the validity of the content being displayed.

9.2.3 Local preferences and token budgeting

Drafting consult letters is inherently iterative, and psychiatrists often develop stable preferences for how much text to generate, how "bold" or "conservative" the AI should be, and whether contextual information should be fully included or de-identified. To avoid forcing the clinician to reconfigure these settings on every visit, `ConsultonPanel` persists key parameters in `localStorage` via a small family of helper functions.

For numerical settings, such as the sampling temperature and maximum token budget, the panel uses:

$$\texttt{readNumberLS} : (\texttt{key}, \texttt{fallback}) \mapsto \mathbb{R}, \tag{9.8}$$

$$\texttt{writeNumberLS} : (\texttt{key}, \texttt{val}) \mapsto \texttt{void}, \tag{9.9}$$

with a subsequent clamping operation

$$\theta \;=\; \mathsf{clamp}\big(\texttt{readNumberLS}(\texttt{key}, \theta_{\text{default}}), \theta_{\min}, \theta_{\max}\big), \tag{9.10}$$

where θ denotes one of the numerical parameters. In the current implementation, the following keys are used:

- `LS_TEMP` (`"CONSULT_TEMP"`) for the sampling temperature, clamped to the interval $[0, 1]$;
- `LS_MAXTOK` (`"CONSULT_MAXTOKENS"`) for the maximum number of tokens the model is allowed to return, clamped to $[64, 8192]$.

Boolean preferences, such as whether to redact contextual data, are stored using:

$$\texttt{readBoolLS} : (\texttt{key}, \texttt{fallback}) \mapsto \{\texttt{true}, \texttt{false}\}, \tag{9.11}$$

$$\texttt{writeBoolLS} : (\texttt{key}, \texttt{val}) \mapsto \texttt{void}, \tag{9.12}$$

with `readBoolLS` interpreting `"1"`, `"true"`, or `"yes"` as `true`. The key `LS_REDACT` (`"CONSULT_REDACT"`) controls whether the context derived from $\text{Audit}(s)$ is passed through the de-identification engine before being sent to the AI provider. When this flag is enabled,

Preference key (localStorage)	Parameter	Clinical and technical role
`"CONSULT_TEMP"`	Sampling temperature	Controls how deterministic or exploratory the model is during consult drafting. Lower values yield more conservative text; higher values allow more varied phrasing and additional details.
`"CONSULT_MAXTOKENS"`	Maximum output tokens	Upper bound on model tokens returned for a single Consulton call. Protects against excessively long responses and enables rough budgeting of API costs.
`"CONSULT_REDACT"`	Context de-identification flag	Boolean toggle indicating whether the contextual JSON derived from Audit(s) should be de-identified before being sent to the AI model. Reflected in the Consulton header as a "Redacted" pill.

Table 39: *Local Consulton preferences stored via `readNumberLS/writeNumberLS` and `readBoolLS/writeBoolLS`. These keys allow the psychiatrist's preferred sampling behaviour and de-identification stance to persist across sessions.*

the "Redacted" pill in the Consulton header provides an immediate visual reminder that the model is operating on a partially masked view of the session.

Table 39 summarises these local preference keys and their clinical roles.

Beyond the raw configuration, `ConsultonPanel` provides a dynamic *token budget visualisation* implemented by the `TokenBudget` component. The underlying heuristic is the helper:

$$\widehat{T}(c) \;=\; \left\lceil \frac{c}{4} \right\rceil, \tag{9.13}$$

where c is the character count of a string and $\widehat{T}(c)$ is the approximate number of tokens. This reflects the empirical observation that for typical English clinical text, one token corresponds to roughly four characters on average.

The component separately estimates:

- $T_{\text{prompt}} = \widehat{T}(|\text{prompt}|)$, the token demand of the clinician's prompt text; and
- $T_{\text{ctx}} \;=\; \widehat{T}(|\text{ctxStr}|)$, the demand of the contextual JSON assembled from Audit(s) by `assembleConsultContext`.

The total estimated demand is then

$$T_{\text{tot}} = T_{\text{prompt}} + T_{\text{ctx}}, \tag{9.14}$$

and the proportion of the user-specified limit M (from `LS_MAXTOK`) is visualised as

$$\pi = \min\left(1,\; \frac{T_{\text{tot}}}{\max(1, M)}\right). \tag{9.15}$$

This proportion π drives the width of the budget bar (`budgetBar`, `budgetFill`) so that the psychiatrist can immediately see when a long prompt or large context is likely to exceed the planned token budget. The meter is purely advisory—the underlying model call still respects the hard limit M—but it fosters a forward-looking awareness of computational cost and latency, which is particularly relevant for busy clinics with constrained consultation time.

9.2.4 Consulton sessions as traceable drafting episodes

While individual Consulton calls produce transient drafts, specialists frequently work on complex reports over multiple sittings: for example, longitudinal risk formulations, medico-legal letters, or detailed transfer summaries. To support this workflow, SynapseCore introduces an explicit *Consulton session* model defined in `ConsultonSessions.ts`. The central type is:

$$\begin{aligned}\texttt{ConsultSession} = \big(&\texttt{id}, \texttt{title}, \texttt{startedAt}, \texttt{endedAt},\\ &\texttt{status}, \texttt{raw}, \texttt{html},\\ &\texttt{model}, \texttt{temperature}, \texttt{maxTokens}\big),\end{aligned} \tag{9.16}$$

where:

- `id` is a unique identifier suitable for use as a React key;
- `title` is a human-readable label (for example "Inpatient consult for Mr. A, 2025-03-12");
- `startedAt` and optional `endedAt` are Unix timestamps in milliseconds delimiting the drafting episode;
- `status` takes values in the finite set $\{\texttt{"running"}, \texttt{"done"}, \texttt{"canceled"}, \texttt{"error"}\}$;
- `raw` contains the Markdown representation of the draft; and
- `html` stores the corresponding sanitised HTML as generated by the pipeline in Equation (9.7);
- `model`, `temperature`, and `maxTokens` record the key AI sampling parameters used for that session.

The life-cycle of a Consulton session can be conceptualised as a small state machine:

$$\texttt{"running"} \xrightarrow{\text{complete}} \texttt{"done"} \xrightarrow{\text{rename/inspect}} \texttt{"done"}, \tag{9.17}$$

with possible early exits to `"canceled"` or `"error"` when the clinician explicitly aborts generation or when the AI call fails. The helper functions `createSession`, `updateSession`, `closeSession`, and `renameSession` implement the corresponding mutations on an in-memory array of `ConsultSession` objects, followed by a `notify()` call that informs subscribers about the new state.

Consumers register via `subscribeSessions`, which attaches a callback that is invoked whenever the sessions array changes. In the UI, this allows a side panel or modal to show:

- the chronological list of saved Consulton sessions;
- the current status of each (running, done, canceled, error);
- timestamps and model configuration; and
- entry points to open, diff, or export a given session.

The `DiffSideBySide` component (from `ConsultonDiff.tsx`) adds an additional layer by computing a line-based diff between two session drafts and rendering them with colour-coded additions and deletions. This is particularly useful when refining medico-legal letters where subtle changes in wording may carry significant implications.

Table 40 summarises the key fields of `ConsultSession` and their clinical interpretation.

9.2.5 Integrated Consulton workflow

Bringing these pieces together, the Consulton subsystem can be seen as a vertical pipeline that begins with an audit-backed clinical context and ends in a versioned set of consult drafts ready for export through the Tools framework. The overall flow is depicted in Figure 23.

Field	Type	Clinical interpretation
`id`	String	Session-local unique identifier linking UI elements to the underlying consult draft.
`title`	String	Human-readable label describing the consult (for example patient initials and date). Facilitates quick retrieval of relevant drafts.
`startedAt`, `endedAt`	Number (ms since epoch)	Timestamps delimiting the drafting episode. Useful for understanding how long complex reports require and for correlating drafting time with clinical workload.
`status`	Enum (`"running"` / `"done"` / `"canceled"` / `"error"`)	Life-cycle state of the consult draft. Supports dashboards or QA views that highlight incomplete or failed drafts.
`raw`	String (Markdown)	Canonical text representation of the consult draft, suitable for editing, diffing, and re-rendering under updated sanitisation rules.
`html`	String (HTML)	Sanitised HTML snapshot used for fast on-screen rendering. Derived deterministically from `raw`.
`model`	String	Identifier of the AI model used during drafting (for example "gpt-4o"). Enables later analysis of how model choice influences document style.
`temperature`	Number	Sampling temperature used when generating the consult text. Recorded for reproducibility and quality analysis.
`maxTokens`	Number	Upper bound on generated tokens. Records the intended length and resource envelope for the draft.

Table 40: *Schema of a Consulton drafting episode as represented by `ConsultSession`. The combination of timestamps, status, and sampling parameters provides a traceable record of how a consult letter or report was generated and refined.*

Within this pipeline, the psychiatrist retains full control over each step: they decide which contextual elements to include (and whether to redact), review the live-rendered output for clinical accuracy and tone, and choose which drafts to keep or discard. At the same time, the underlying code provides strong guarantees about safety (through sanitisation), reproducible configuration (through stored preferences and session metadata), and auditability (through the linkage to Audit(s^*)). As a result, Consulton serves as a high-trust environment for AI-assisted documentation within the broader SynapseCore digital psychiatry workbench.

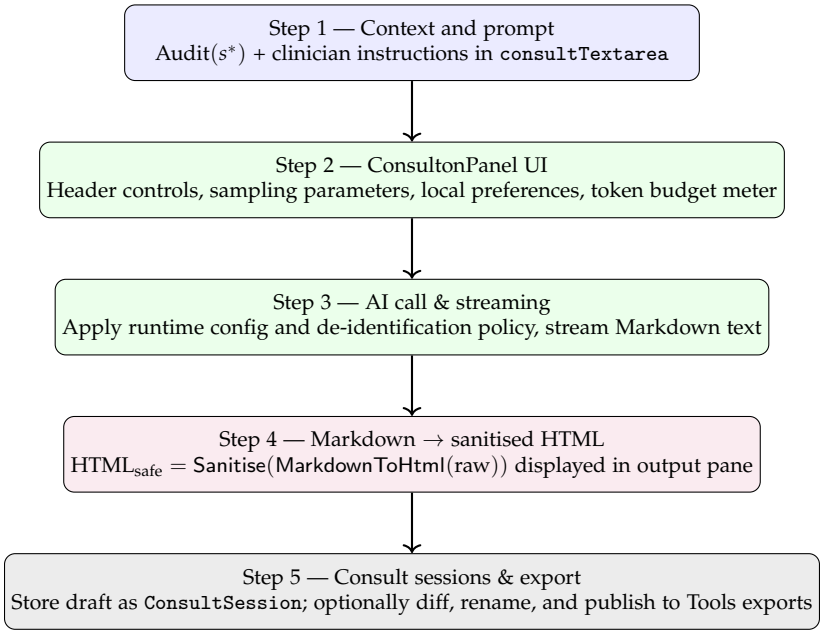


Figure 23: *Integrated Consulton workflow. An audit-backed context and clinician prompt enter the ConsultonPanel UI, which manages sampling parameters, local preferences, and token budgeting. The AI call returns Markdown text that is converted to sanitised HTML, displayed in the output pane, and stored as a* `ConsultSession` *that can be diffed, renamed, and exported via the Tools framework.*

9.3 Export Recipes

The final step in the SynapseCore documentation pipeline is to transform the audit-backed context and Consulton drafts into concrete artefacts that can be stored, transmitted, or analysed outside the application. This step is encapsulated by the *export recipes* under `centerpanel/Tools/exportRecipes/`, together with the generic assembly and Markdown utilities in `Tools/lib/assemble.ts`, `Tools/lib/markdown.ts`, and `Tools/lib/scope.ts`. From a clinical perspective, each recipe corresponds to a recognisable product (e.g. a progress note, a consult letter, or a tabular export of measurement scores). From a software perspective, each recipe is a pure transformation from a typed *scope* object into one or more serialised formats (Markdown, HTML, CSV, JSON) that can be enqueued in the export inbox and copied to external systems.

9.3.1 Recipe abstractions and type-level design

All export recipes share a common type-level abstraction. Let $\mathcal{S}$ denote the set of permissible scope objects (constructed as described in Section 9.3.2 below), and let $\mathcal{F}$ denote the set of supported output formats (`md` for Markdown, `html`, `csv`, `json`). At the code level, a recipe r is an object of the form:

$$\texttt{ExportRecipe} = \big(\texttt{id}, \texttt{label}, \texttt{format}, \texttt{build}, \texttt{postProcess}, \texttt{meta}\big), \tag{9.18}$$

where:

- `id` is a stable identifier used by the Tools framework and persisted in the export inbox;
- `label` is a clinician-facing string such as “Session progress note” or “Risk-focused letter”;
- `format` is an element of $\mathcal{F}$ indicating the primary serialisation of the document;

Recipe family	Primary formats	Typical clinical use
Note exports	Markdown, HTML	Standard progress notes summarising mental state examination, risk, interventions, and plan for a single session.
Letters	Markdown, HTML	Detailed consult or handover letters integrating findings from multiple sessions and structured flows, often intended for other clinicians or services.
Data exports	CSV, Markdown tables, JSON	Structured output of measurement-based care scores, session timelines, and flow outcomes for research, audit, or quality-improvement pipelines.

Table 41: *High-level grouping of export recipes. Each recipe family uses the same underlying scope construction but emphasises different sections of the clinical record and serialisation formats.*

- `build` is a pure function $\mathcal{S} \to \mathsf{DocTree}$ that maps a scope into a structured document tree;
- `postProcess` is an optional hook that can, for example, inject boilerplate headers or ensure the presence of required sections;
- `meta` stores auxiliary information such as whether the recipe expects AI-generated paragraphs, whether it should respect de- identification flags, and how it should appear in the Tools UI.

The concrete recipes in `exportRecipes/consult.ts` and related files conform to this shape. For example, a narrative consult recipe builds a DocTree consisting largely of headings and paragraphs, whereas a data export recipe yields a tree dominated by tables and code fences.

To maintain conceptual clarity, recipes are grouped into three broad families:

- **Note recipes**, which produce intra-session documentation such as progress notes and risk summaries from $\mathrm{Audit}(s)$;
- **Letter recipes**, which generate outward-facing documents (e.g. consult letters, transfer summaries) that may combine multiple sessions and include prosaic explanation of diagnosis, risk, and plan;
- **Data export recipes**, which output structured CSV, Markdown tables, or JSON objects representing scale scores, flows, and timer events for subsequent analysis.

A simplified overview of these families is given in Table 41.

9.3.2 Scope and assembly of export contexts

Recipes operate on an intermediate *scope* object that abstracts away the details of where data originated. The construction of this scope is handled by `Tools/lib/scope.ts` and `Tools/lib/assemble.ts`. Let $\mathrm{Audit}(s)$ denote the session-level audit object introduced in Equation (8.11), and let ConsultSession denote an individual Consulton drafting episode as described in Section 9.2. A scope object for a single-session export is defined as:

$$\begin{aligned}\mathsf{Scope}(s^*) = \big(&\text{patientInfo},\ \text{sessionInfo},\ \mathcal{T}_{s^*},\ \mathcal{F}_{s^*},\\ &\mathcal{N}_{s^*},\ \mathcal{M}_{s^*},\ \mathcal{C}_{s^*}\big),\end{aligned} \tag{9.19}$$

where:

- patientInfo encodes patient-level demographics and identifiers (often de-identified in the export context);

- sessionInfo summarises the selected consultation (date, setting, clinician role, and tags such as "crisis" or "follow-up");
- $\mathcal{T}_{s^*}$, $\mathcal{F}_{s^*}$, $\mathcal{N}_{s^*}$, $\mathcal{M}_{s^*}$ are the timeline, flows, notes, and MBC components of Audit(s^*);
- $\mathcal{C}_{s^*}$ contains zero or more linked ConsultSession objects that have been explicitly associated with the session and are eligible to contribute paragraphs to the final document.

The function `buildScopeFromAudit` (in `scope.ts`) performs the mechanical work of assembling this object:

B1 Pull session metadata and patient data from the global store;
B2 Normalise timeline and flow outcomes into arrays of small, reusable `Slice` or `Fact` objects (for example "agitation present, moderate severity, de-escalated with verbal techniques");
B3 Attach the most recent MBC result set $\mathcal{M}_{s^*}$ and derive severity bands and risk grades;
B4 Optionally inject excerpts from one or more ConsultSession drafts if the recipe requests them via metadata (for example, a consult-letter recipe may pull in the latest accepted Consulton narrative for the "History" section).

For multi-session or longitudinal exports, a higher-order scope constructor aggregates across a finite set of sessions $\{s_1, \ldots, s_n\}$, producing a scope of the form:

$$\mathsf{Scope}^{\text{multi}} = \big(\text{patientInfo},\ \{\mathsf{Scope}(s_i)\}_{i=1}^{n}\big), \tag{9.20}$$

from which recipes can derive trajectories of scores or evolving risk formulations.

9.3.3 Markdown generation as canonical representation

Although the export layer supports several output formats, Markdown is treated as the canonical textual representation. The module `Tools/lib/markdown.ts` defines a small domain-specific language for constructing Markdown in a composable and testable way. At its core is a DocTree abstraction built from node types such as Heading, Paragraph, List, Table, and CodeFence.

Given a tree $D \in \mathsf{DocTree}$, the renderer MkMarkdown is a pure function:

$$\mathsf{MkMarkdown} : \mathsf{DocTree} \longrightarrow \mathsf{Markdown}, \tag{9.21}$$

which is implemented by a depth-first traversal that emits Markdown lines and joins them with newline characters. Helper functions in `markdown.ts` ensure that recipes rarely manipulate raw strings directly; instead they use small combinators such as:

- `h(level, text)` to create a heading node,
- `p(text)` to create a paragraph node,
- `ul(items)` to create unordered lists, and
- `table(headers, rows)` to create Markdown tables with the correct separator row.

This design has two advantages. First, it reduces the risk of syntactic errors (for example missing header underlines or misaligned table separators), which would otherwise lead to inconsistent rendering across Markdown viewers. Second, it enables unit tests that operate on the trees rather than on raw strings, simplifying regression testing when recipes evolve.

Once Markdown has been produced, other formats are derived via simple transformations. For example, HTML is generated by applying the same MarkdownToHtml + Sanitise pipeline discussed in Section 9.2.2, whereas CSV and JSON exports serialise specific views of the scope (such as rows of $[\text{date}, \text{scale}, \text{score}, \text{severity}]$).

9.3.4 Multi-format serialisation and data exports

Many modern psychiatric services require both narrative documentation and structured data feeds for dashboards, quality audits, or research projects. The export recipes therefore extend beyond narrative Markdown to include CSV and JSON views of the same underlying scope.

Let D be the document tree produced by a recipe, and let $\pi_{\text{data}}(D)$ denote a projection that extracts only those elements corresponding to structured metrics (for example MBC scores, risk grades, or lap durations from the timer). The serialisers in `assemble.ts` implement functions of the form:

$$\mathsf{toMarkdown}(D) = \mathsf{MkMarkdown}(D), \tag{9.22}$$

$$\mathsf{toCsv}(D) = \mathsf{CsvEncode}\big(\pi_{\text{data}}(D)\big), \tag{9.23}$$

$$\mathsf{toJson}(D) = \mathsf{JsonEncode}\big(\pi_{\text{data}}(D)\big), \tag{9.24}$$

where CsvEncode and JsonEncode are straightforward formatters that respect column headers and value types.

Data-oriented recipes in `exportRecipes` use these serialisers to produce, for example:

- CSV files containing one row per measurement occasion, with columns for patient pseudonym, date, scale, raw score, severity band, and risk-grade mapping;
- JSON objects encoding per-session feature sets (including timer laps, flow outcomes, and MBC scores) suitable for ingestion into analytic notebooks or ETL pipelines;
- Markdown tables summarising key metrics for human-readable reports in quality-improvement committees.

Narrative recipes typically produce only Markdown (and derived HTML), but they share the same assembly logic and can therefore be extended to emit structured attachments if required.

9.3.5 Vertical export-recipe pipeline

Figure 24 illustrates the full export pipeline as a vertical sequence of transformations: from a scoped context built over $\text{Audit}(s)$ and Consulton drafts, through recipe-specific document construction, to format-specific serialisation and finally to the export inbox managed by `exportInbox.ts` and `ExportBar.tsx`.

By enforcing this layered design, SynapseCore ensures that all exportable artefacts are (i) grounded in a typed scope derived from the same session audit record, (ii) constructed via composable and testable document trees, and (iii) serialised through a small, well-understood set of formatters. For the practising psychiatrist, this translates into a predictable, low-friction experience when generating notes, letters, or data extracts, with clear separation between clinical judgement and mechanical formatting concerns.

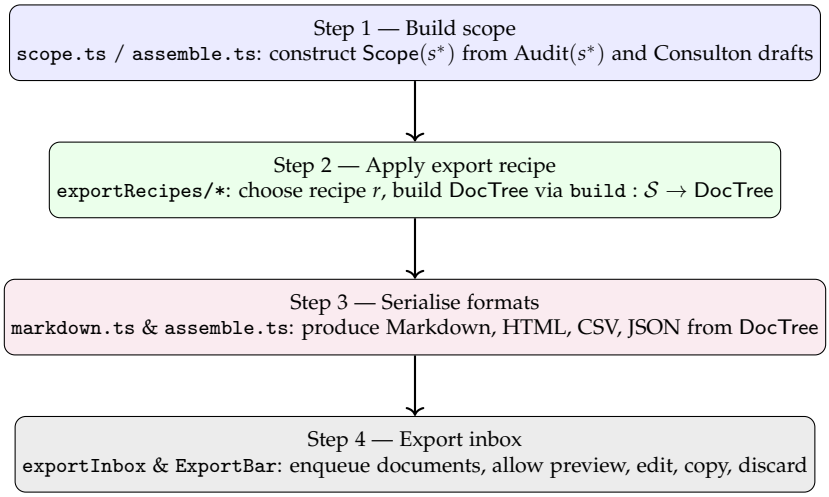


Figure 24: *Vertical export-recipe pipeline in SynapseCore. A scope object built from the session audit and Consulton drafts is passed through a recipe-specific builder to create a* DocTree*. Generic serialisers then produce Markdown, HTML, CSV, or JSON, which are materialised as export entries in the Tools inbox where the psychiatrist can review and dispatch them.*

9.4 Virtual Lists and Performance

As the SynapseCore workbench matures into a long-term clinical tool, the number of items displayed in the Tools and Consulton views can become very large. A single patient may accumulate hundreds of sessions, each associated with multiple exports (notes, letters, data extracts) and several Consulton drafts. Rendering all of these entries as plain React lists would incur a quadratic burden on the browser: the DOM tree would grow linearly with the number of entries, and each scroll or filter operation would require the layout engine to recompute style and position for every item.

To avoid this, the Tools module employs a dedicated `VirtualList` component (`Tools/components/VirtualList.tsx`) that implements *virtual scrolling*. Instead of rendering all N logical items, the component renders only those entries that are visible in the viewport plus a small buffer above and below. This keeps the number of live DOM nodes bounded by a constant independent of N, yielding a predictable and responsive user experience even when the underlying list is large.

9.4.1 Viewport model and index mapping

The behaviour of `VirtualList` can be described using a simple geometric model of scrolling. Consider a vertical list of N items, each with fixed height $h > 0$ in pixels. Let H denote the height of the scrolling viewport (the visible region of the list), and let $y \geq 0$ be the current vertical scroll offset from the top of the list.

The logical index range of items fully or partially visible at offset y is:

$$\begin{aligned} i_{\min}(y) &= \max\Big(0,\ \Big\lfloor \frac{y}{h} \Big\rfloor\Big), \\ i_{\max}(y) &= \min\left(N-1,\ \left\lfloor \frac{y+H}{h} \right\rfloor\right). \end{aligned} \tag{9.25}$$

To mitigate the perceptual artefact of items popping into view exactly at the viewport bound-

ary, `VirtualList` introduces an *overscan* parameter $k \in \mathbb{N}$, which pads the range:

$$\begin{aligned} i^*_{\min}(y) &= \max\bigl(0,\ i_{\min}(y) - k\bigr), \\ i^*_{\max}(y) &= \min\bigl(N-1,\ i_{\max}(y) + k\bigr). \end{aligned} \tag{9.26}$$

Only items with indices in the closed interval $[i^*_{\min}(y), i^*_{\max}(y)]$ are actually rendered as DOM nodes. The vertical position of the first rendered item is implemented via a `transform: translateY(...)` style or an equivalent spacer element so that the scroll bar reflects the full logical height Nh, even though only a subset of items is present in the DOM.

The cardinality of the rendered window is at most:

$$M(y) = i^*_{\max}(y) - i^*_{\min}(y) + 1 \ \le\ \left\lceil \frac{H}{h} \right\rceil + 2k, \tag{9.27}$$

which is independent of N. In the Tools context, typical values might be $H \approx 600$ px, $h \approx 44$ px, and $k = 4$, yielding $M(y) \lesssim 21$ visual rows regardless of whether there are 50 or 5000 logical items.

9.4.2 Component interface and rendering pipeline

The `VirtualList` component is designed as a generic container that can render any list of items, not only exports. Its TypeScript interface can be schematically summarised as:

$$\begin{aligned} \texttt{VirtualListProps} = \bigl(&\texttt{items}, \texttt{rowHeight}, \texttt{overscan}, \\ &\texttt{renderItem}, \texttt{className}, \texttt{onScroll}\bigr), \end{aligned} \tag{9.28}$$

where:

- `items` is an array of arbitrary values (for example export entries or Consulton sessions);
- `rowHeight` is the fixed height h;
- `overscan` is the padding k used in Equation (9.26);
- `renderItem` is a pure function that maps $(\texttt{item}, \texttt{index})$ to a React node;
- `className` allows the calling component to apply its own CSS styling (for example `toolsList`, `exportList`);
- `onScroll` is an optional callback for higher-level analytics or synchronisation.

Internally, the component maintains the current scroll offset y in a `useState` hook and attaches an `onScroll` handler to a container `div`. On each scroll event, the handler:

V1 reads `event.currentTarget.scrollTop` into y;
V2 computes $i^*_{\min}(y)$ and $i^*_{\max}(y)$ via Equations (9.25) and (9.26);
V3 slices the `items` array to obtain the window $\{x_i\}_{i=i^*_{\min}}^{i^*_{\max}}$;
V4 computes a vertical offset $\delta(y) = i^*_{\min}(y)\,h$ and applies it as a transform to the inner container;
V5 calls `onScroll` if provided.

Only the sliced window is mapped through `renderItem`, which keeps the render cycle inexpensive even when the total list size is large.

At the DOM level, the structure can be conceptualised as:

- an outer `div` with fixed `height: H` and `overflow-y: auto`;
- an inner `div` with `height: N h` and a `transform` equal to $\delta(y)$;
- child nodes corresponding only to items with indices in the current window.

This pattern is particularly effective in the Tools context where export lists and note histories are mostly read-only; interactions such as clicking an item or opening a context menu are local to the rendered window and do not require knowledge of off-screen elements.

9.4.3 Complexity analysis and practical performance

From a complexity standpoint, the advantage of virtualisation is immediate. If a naive list renders all N items, then for each scroll event the cost of layout and paint is approximately $O(N)$, and the memory usage is also $O(N)$ with a large constant factor (each item may contain icons, buttons, and nested elements). By contrast, with `VirtualList` the number of rendered items is bounded by $M(y)$, so both time and space cost per scroll event are $O(1)$ with respect to N:

$$\text{Time per scroll} \approx c_1\, M(y) \le c_1 \left(\left\lceil \frac{H}{h} \right\rceil + 2k \right), \tag{9.29}$$

$$\text{DOM size} \approx c_2\, M(y), \tag{9.30}$$

for constants c_1, c_2 determined by the complexity of individual list items. In practice, this translates into:

- stable frame rates when scrolling through hundreds or thousands of exports or notes;
- reduced garbage collection pressure, since fewer React elements and DOM nodes are created and destroyed;
- lower energy consumption on mobile devices and laptops, which is non-trivial in busy outpatient settings with limited charging opportunities.

Clinically, the practical effect is that long-term patients with multi-year histories remain browsable without perceptible lag. The psychiatrist can rapidly scan back through prior exports or note snapshots to contextualise current risk assessments, without experiencing the “sticky” scroll behaviour that often accompanies large DOM trees.

9.4.4 Parameter choices and tuning

The parameter triplet (h, H, k) governs the trade-off between perceived smoothness and resource usage. Table 42 summarises their roles and typical values in the SynapseCore Tools context.

In the implementation, the Tools lists also leverage React memoisation (`React.memo`) for individual row components, so that even within the virtual window only those items whose props have changed are re-rendered. This is particularly relevant for export entries whose status field (`pending`, `edited`, `copied`, `discarded`) may update independently of scroll events.

9.4.5 Vertical schema of virtualised rendering

Figure 25 summarises the virtualised rendering process as a vertical schema: an abstract list of N logical items at the top, a scrolled viewport in the middle, and the limited set of DOM nodes realised at the bottom.

By encapsulating this behaviour in a dedicated `VirtualList` component, SynapseCore ensures that any future module needing to display large collections (for example longitudinal MBC plots or extensive audit logs) can re-use the same performance-aware pattern. This preserves the responsiveness of the workbench as it scales to the realistic data volumes encountered in modern psychiatric practice.

Parameter	Typical value	Effect on behaviour
Item height h	$\approx$ 40–48 px	Determines how many items fit in the viewport. Too small values increase information density but may degrade readability; too large values reduce the information visible at once.
Viewport height H	$\approx$ 540–640 px	Controlled by surrounding layout. Larger H increases $M(y)$ but improves context when scanning lists; smaller H reduces immediate context but may be necessary on low-resolution devices.
Overscan k	3–6 items	Balances smoothness against cost. Higher k reduces the chance of visible pop-in at the edges of the viewport but slightly increases the number of rendered items; very low k maximises performance but may produce perceptible boundary artefacts.

Table 42: *Key parameters of the `VirtualList` component. The chosen defaults keep the number of rendered items small while avoiding visual artefacts at the viewport boundary.*

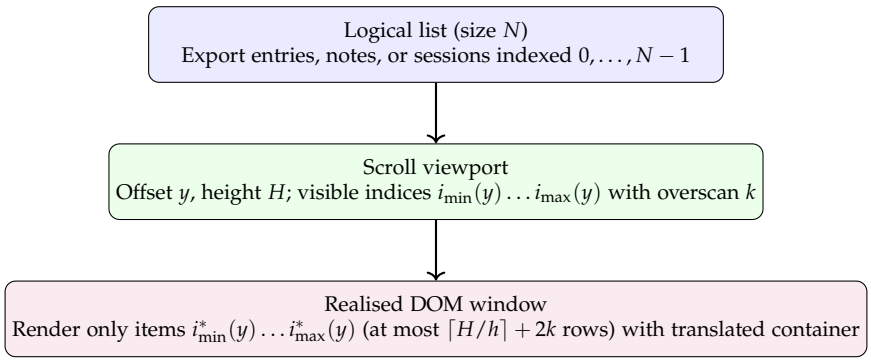


Figure 25: *Virtual list rendering in the Tools module. The full logical list of size N is never rendered in its entirety. Instead, the scroll viewport determines a narrow index window which is mapped to a small set of DOM nodes via `VirtualList`.*

10 User Interface, Layout Shells, and Synapse Theme

The SynapseCore workbench is presented to the clinician through a single-page React application whose composition root is the pair `AppRoot.tsx+App.tsx`. These components, together with the status bar, theme provider, modal stack, and AI assistant root, define the *application shell*: the stable frame within which all psychiatric tools, flows, timers, and documentation surfaces are embedded.

Formally, if we let $\mathcal{F}$ denote the set of feature surfaces (e.g. Psychiatry Workbench, Projects, Settings) and $\mathcal{M}$ the set of global overlays (modals, AI assistant, toasts), then the shell at time t can be described as a triple

$$\mathsf{Shell}_t = \big(\mathsf{StatusBar}_t,\ \mathsf{Viewport}_t \in \mathcal{F},\ \mathsf{Overlays}_t \subseteq \mathcal{M}\big), \tag{10.1}$$

with the invariant that $\mathsf{StatusBar}_t$ and the theme remain globally consistent regardless of which feature surface is active or which modal stack is currently visible. The following subsections detail how `AppRoot.tsx`, `App.tsx`, and related components realise this structure.

10.1 Application Shell

10.1.1 Composition root and routing

At the entry point of the web application, the React tree is mounted inside `index.tsx` using the canonical pattern:

```
createRoot(container).render( <BrowserRouter><AppRoot /></BrowserRouter> ).
```
(10.2)

Here, `BrowserRouter` from `react-router-dom` provides the HTML5 history interface and URL-to-route mapping, while `AppRoot.tsx` is responsible for interpreting the current route and selecting the appropriate feature surface. We may formalise the router as a function

$$\mathsf{Route} : \mathcal{U} \to \mathcal{F}, \tag{10.3}$$

where $\mathcal{U}$ is the set of URL pathnames and $\mathcal{F}$ the set of feature components. In the current implementation, Route covers at least:

- the default psychiatry workbench route (for example `/`),
- project-related routes (for example `/project/:id`),
- auxiliary pages such as settings or debug views.

Within `AppRoot.tsx`, these mappings are expressed as a tree of `<Routes>` and `<Route>` elements. Regardless of which branch matches, the leaf route renders `<App />` as its main element, so that all feature surfaces share the same surrounding shell.

We can therefore regard `AppRoot` as an *adapter* that converts routing state $u_t \in \mathcal{U}$ into a feature component $F_t \in \mathcal{F}$, and passes this into `App` as the *current viewport*. Equation (10.1) then becomes, at runtime,

$$\mathsf{Shell}_t = \mathsf{App}\big(F_t,\, \theta_t\big), \tag{10.4}$$

where θ_t summarises global configuration such as the active theme and AI provider settings.

10.1.2 Status bar and global surfaces

The `StatusBar` component is rendered at the top of `App.tsx` and is conceptually independent of which feature is active. It exposes:

- application identity and version (for example “SynapseCore” plus a semantic version or commit hash);
- high-level connectivity indicators (online/offline state, AI provider availability);
- access points for global actions (opening the AI settings modal, toggling theme, showing keyboard shortcuts).

At a formal level, the status bar is a function of global app state Γ_t only:

$$\mathsf{StatusBar}_t = \mathsf{StatusBar}(\Gamma_t), \tag{10.5}$$

where Γ_t is provided by global React context hooks (for example `useAppConfig`, `useAiConfigStore`). Crucially, Γ_t does not depend on the local state of any center panel feature; this ensures that changes to a timer or flow do not trigger unnecessary re-renders of the status bar.

In the JSX structure of `App.tsx`, `<StatusBar />` typically appears as the first child within the main `<div className="appRoot">`, followed by the central layout shell and the overlay stack. This means that, visually, the status bar is always present at the top of the viewport, reinforcing a sense of continuity as the psychiatrist navigates between different tools.

10.1.3 Theme provider and layout slots

The visual identity of the workbench—colour palette, typography, spacing, elevation, and motion—is centralised in `AppThemeProvider.tsx`. At the React level, this component wraps the entire application:

$$\texttt{<AppThemeProvider><AppShell /></AppThemeProvider>}, \tag{10.6}$$

injecting theme variables via React context and CSS custom properties. If we let Θ denote the space of available themes (for example light, dark, and neutral clinical variants), then at any time the provider selects a theme $\theta_t \in \Theta$ based on user settings and system preferences (e.g. `prefers-color-scheme`). All descendant components read θ_t through lightweight hooks (for example `useTheme`) or directly via CSS variables.

From a layout perspective, `App.tsx` divides the remaining vertical space beneath the status bar into three principal slots:

$$\mathsf{Viewport}_t = \big(\mathsf{LeftRail}_t,\ \mathsf{CenterPanel}_t,\ \mathsf{RightDock}_t\big), \tag{10.7}$$

where:

- $\mathsf{LeftRail}_t$ typically contains navigation, patient selection, or high-level mode switches;
- $\mathsf{CenterPanel}_t$ hosts the main active surface (for example the psychiatry workbench, flows, or the Tools view);
- $\mathsf{RightDock}_t$ hosts contextually relevant panels, such as AI explanations, MBC detail views, or configuration inspectors.

These slots are styled via a combination of global CSS (e.g. the `appShell` class) and feature-specific modules. The theme provider ensures that colours, fonts, and border radii remain aligned across slots, so that timers, flows, and tools feel like parts of a single, coherent instrument rather than disconnected widgets.

10.1.4 Modal stack and AI assistant root

Beyond the main shell, `App.tsx` also composes a number of top-level modals whose visibility is controlled by global state:

- `NewProjectModal`: creation of new workspaces or projects, including choices of templates and access to example datasets;
- `PsychiatryModal`: high-level psychiatry-specific settings, such as which flows are enabled, default MBC scales, and safety configuration;
- `WelcomeModal`: an onboarding surface for first-time users or after major upgrades, often describing new features or changes in the AI stack;
- AI settings modal(s), exposing runtime AI configuration such as provider selection, model choice, and sampling presets;
- other diagnostics or support modals (feature flags, debug panels).

Collectively, these modals constitute the overlay set $\mathsf{Overlays}_t$ in Equation (10.1). Each modal is conditionally rendered based on a boolean in global state, often persisted in `localStorage` so that preferences survive page reloads.

In addition, `App.tsx` mounts the global AI assistant root via `components/ai.ts`. This component is responsible for:

- providing a floating AI panel or dock that can be summoned from any feature surface;
- multiplexing requests across different AI use cases (for example ad-hoc clinical questions, code explanations, or contextual summarisation of session content);

Component	Location	Clinical and technical role
`AppRoot`	`AppRoot.tsx`	Integrates with `BrowserRouter` to map URLs to feature surfaces. Ensures that the same application shell wraps all major tools and views.
`App`	`App.tsx`	Core application shell. Composes `StatusBar`, `AppThemeProvider`, layout slots (left rail, center panel, right dock), modal stack, and AI assistant root.
`StatusBar`	`components/StatusBar.tsx`	Displays app identity, connectivity, and global actions. Provides a stable cognitive anchor across all workflows.
`AppThemeProvider`	`AppThemeProvider.tsx`	Provides the Synapse theme via React context and CSS variables; coordinates light/dark/neutral modes and typography for a coherent visual identity.
Modal components	`components/modals/*`	Top-level overlays (`NewProjectModal`, `PsychiatryModal`, `WelcomeModal`, AI settings). Surface configuration and onboarding workflows without disturbing the underlying session state.
`AiAssistantRoot`	`components/ai.ts`	Global AI assistant entry point. Manages streaming conversations, provider settings, and contextual prompts, accessible from any feature surface.

Table 43: *Principal components of the SynapseCore application shell and their roles in composing a stable, theme-aware user interface around the psychiatry workbench.*

- integrating with the same runtime configuration layer used by Consulton and the MBC explanation engine, so that provider and model choices remain consistent.

From a systems perspective, the AI assistant root is a specialised overlay in $\mathsf{Overlays}_t$ whose internal routing distinguishes between psychiatry-oriented prompts and developer-oriented prompts, but whose existence is independent of the current $\mathsf{Viewport}_t$.

Table 43 summarises the main components involved in the application shell and clarifies their roles from both a clinical and technical standpoint.

10.1.5 Vertical schema of the application shell

The overall control flow of the application shell can be visualised as a vertical schema, shown in Figure 26. URL state and theme configuration enter at the top, pass through `AppRoot` and `App`, and result in a concrete layout plus overlays at the bottom.

By isolating routing, theming, and overlays into well-defined components, the SynapseCore shell supports substantial clinical and technical evolution without destabilising the core user experience. New flows, tools, or AI capabilities can be added as feature surfaces within the existing layout, while the psychiatrist continues to interact with a familiar status bar, theme, and modal behaviour across the entire digital psychiatry workbench.

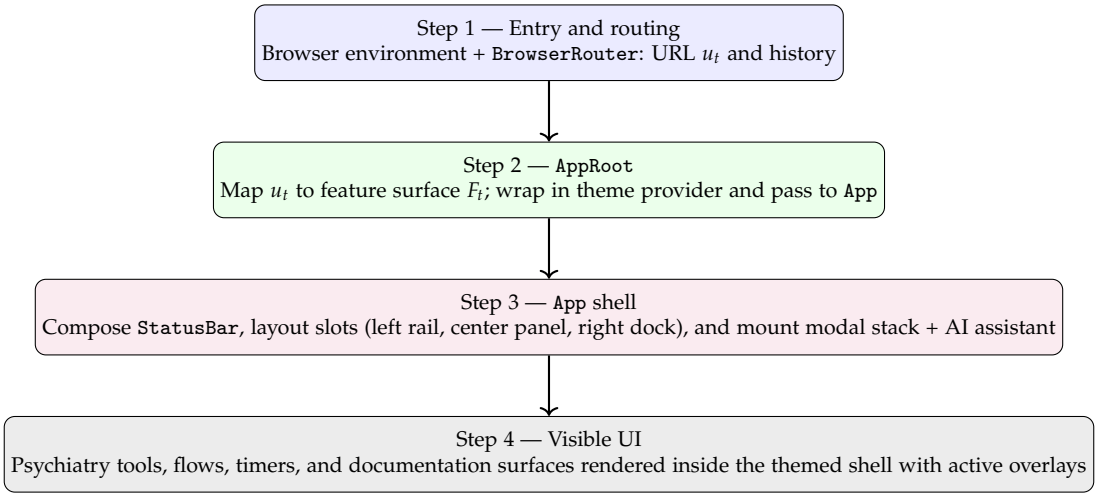


Figure 26: *Vertical schema of the SynapseCore application shell. URL and configuration state flow from the browser environment through* `AppRoot` *and* `App`*, resulting in a themed layout with status bar, feature surfaces, and overlays.*

10.2 Theme Context and Providers

The user interface of SynapseCore relies on a deliberately layered theming architecture that separates (i) stable design tokens, (ii) application-level theme variants (light, dark, neutral), and (iii) the Synapse-specific glassmorphism skin used for the psychiatry workbench. This layering is realised in React through two nested providers:

1. the `ThemeProvider` exported from `src/contexts/ThemeContext.tsx`, which owns the active theme variant and exposes design tokens, and
2. the `AppThemeProvider` defined in `src/app/AppThemeProvider.tsx`, which injects the Synapse theme (`synapseTheme`) and global CSS variables used by higher-level components.

Both providers are implemented with `styled-components`, so that every component can consume a typed `theme` object without manual wiring.

10.2.1 Active theme state and `ThemeContext`

The `ThemeContext` module defines a small, typed interface that encapsulates all theme-related state and actions:

```
interface ThemeContextType {
  theme: Theme;
  themeName: ActiveThemeName;
  setTheme: (theme: ThemeName) => void;
  toggleTheme: () => void;
  designTokens: typeof DESIGN_TOKENS;
  applyThemeTransition: () => void;
}
```

The corresponding type-level representation in `src/styles/theme.ts` constrains the space of admissible theme variants:

```
export type ThemeName = 'light' | 'dark' | 'neutral' | 'auto';
```

```
export type ActiveThemeName = Exclude<ThemeName, 'auto'>;
```

At runtime, the context maintains a pair

$$(\theta_t, \tau_t) \in \mathcal{T} \times \mathcal{N}$$

where $\mathcal{N} = \{\texttt{light}, \texttt{dark}, \texttt{neutral}\}$ is the finite set of active theme names and $\mathcal{T}$ is the space of concrete `Theme` objects. The map

$$\Theta : \mathcal{N} \to \mathcal{T}, \qquad \tau \mapsto \Theta(\tau)$$

is implemented by the `themes` object in `src/styles/theme.ts`, which assembles lighting, color ramps, glassmorphism presets, typography, and spacing into an internal `Theme` value:

$$\theta_t = \Theta(\tau_t).$$

Thus, at any time t the UI part of the global state S_t^{ui} (cf. the discussion in Section 3) contains the current theme name τ_t and its realised theme object θ_t.

The context component created in `src/contexts/ThemeContext.tsx` initialises this state using `useState` and `useEffect`. On mount, it typically:

- inspects persisted preferences (for example, in `localStorage`),
- optionally consults the operating system or browser light/dark preference, and
- chooses an initial $\tau_0 \in \mathcal{N}$ consistent with both.

Whenever `setTheme` is invoked, the state (θ_t, τ_t) is updated by recomputing $\theta_{t+1} = \Theta(\tau_{t+1})$ and the new choice is persisted. This ensures that the workbench honours a clinician's preference across sessions without leaking theme details into individual components.

The `toggleTheme` action exposes a simple cyclic transition function over the active theme names. At the type level this is an endomorphism

$$\text{toggle} : \mathcal{N} \to \mathcal{N},$$

implemented in `src/App.tsx` by the `ThemeToggle` component. In concrete terms, the logic follows the pattern

$$\text{toggle}(\texttt{light}) = \texttt{dark}, \quad \text{toggle}(\texttt{dark}) = \texttt{neutral}, \quad \text{toggle}(\texttt{neutral}) = \texttt{light}.$$

The `ThemeToggle` renders a compact button in the header, with a `lucide-react` icon (`Sun`, `Moon`, or `Monitor`) chosen as a function of τ_t. The accessible label

```
aria-label = "Switch to ...  theme"
```

is computed from the *next* theme, so that screen-reader users are always told what will happen before activation. Clinically, this makes it easy to adapt the visual environment to the setting: a bright palette for teaching rounds and demonstrations, a dark palette for evening clinics, and a more neutral, soft-contrast variant for sensitive consultations.

The `designTokens` field in the context simply re-exports the `DESIGN_TOKENS` constant from `src/constants/design.ts`. This constant aggregates Semantic UI primitives—spacing scales, color systems, radii, elevation, blur, and animation timings—into a single, typed object. Domain components (for example, the risk strip, timer modal, or flow shells) are encouraged to derive layout from these tokens rather than hard-coding pixel values. In formal terms, each visual component implements a function

$$f : \mathcal{T} \times \mathcal{D} \to \mathcal{L},$$

Table 44: Active theme variants in `ThemeContext` and their intended clinical usage. Column widths are bounded to avoid overflow, and textual descriptions are wrapped within cells.

Theme name	Visual characteristics	Typical use in SynapseCore
`light`	High-luminance background with clear surface hierarchy, neutral grays, and accent colors tuned for print-like legibility.	Default for teaching labs, supervision sessions, and environments with strong ambient light where paper-like contrast is desirable.
`dark`	Low-luminance background, reduced glare, and accent colors designed to remain WCAG-compliant against dark surfaces.	Evening clinics, on-call work, and scenarios where bright backgrounds would be fatiguing or distracting for patient and clinician.
`neutral`	Soft-contrast, low-saturation surfaces with careful separation between primary content and chrome elements; visually closer to the Synapse clinical skin.	Sensitive consultations, long-form note writing, and situations where a calmer, less stylised palette is preferred without sacrificing structure.

where $\mathcal{D}$ denotes the space of design tokens and $\mathcal{L}$ the space of layout parameters (margins, paddings, radii). By treating `DESIGN_TOKENS` and the current theme θ_t as inputs, f remains stable across theme changes and keeps clinical affordances (hit-area sizes, type scales) invariant.

Finally, `applyThemeTransition` implements a short, global transition animation. Internally, it typically:

1. adds a CSS class on `document.documentElement` that enables smooth transitions for color-related properties, and
2. removes the class again after a fixed interval (for example, 350 ms) using `setTimeout`.

If we denote the global DOM state by Ω_t, this can be thought of as a bounded operator

$$\Xi : \mathcal{N} \times \Omega_t \to \Omega_{t+1}$$

that preserves layout while allowing light/dark changes to occur without sudden flashes. For clinicians, this reduces visual fatigue when switching themes mid-session.

10.2.2 Composed providers and Synapse theme injection

At the React tree level, theme provisioning is implemented through two nested `styled-components` providers. The outer provider is the `ThemeProvider` exported from `src/contexts/ThemeContext.tsx` and aliased locally as `ThemeProvider` in `src/App.tsx`. The top-level `App` component wraps the application as follows:

```
return (
  <ErrorBoundary>
    <ThemeProvider>
      <AppThemeProvider>
        <GlobalStyles />
        <Router>
          {/* Main SynapseCore UI */}
        </Router>
      </AppThemeProvider>
    </ThemeProvider>
  </ErrorBoundary>
);
```

The outer `ThemeProvider` is responsible for choosing θ_t based on the active theme name τ_t and exposing both via React context. It also brings `DESIGN_TOKENS` into scope. The inner `AppThemeProvider` is defined in `src/app/AppThemeProvider.tsx`:

```
export default function AppThemeProvider({ children }: PropsWithChildren) {
  return (
    <ThemeProvider
      theme={(outer) => ({
        ...(outer as any),
        synapse: synapseTheme,
        focusRing: synapseTheme.focusRing,
      })}
    >
      <GlobalSynapseStyles />
      {children}
    </ThemeProvider>
  );
}
```

Here, the `ThemeProvider` comes directly from `styled-components`. Instead of replacing the existing theme, it uses a *theme function* that merges the outer theme object with the Synapse clinical skin:

$$\tilde{\theta}_t = \mathrm{merge}(\theta_t, \sigma),$$

where σ is the `synapseTheme` imported from `src/theme/synapse.ts`. The merged theme $\tilde{\theta}_t$ preserves all fields supplied by `ThemeContext` (colors, typography, spacing, glass presets), but extends it with:

- a `synapse` namespace containing glassmorphism-friendly background, border, and text colors (for example, `bg900`, `surface800`, `gold500`, `text100`), and
- a `focusRing` helper that returns a CSS snippet implementing a high-contrast focus outline consistent with the Synapse visual identity.

The `GlobalSynapseStyles` component, defined in `src/theme/GlobalSynapseStyles.ts`, injects a set of CSS custom properties (`-syn-bg-900`, `-syn-surface-800`, etc.) onto the `:root` element and normalises scrollbars, preformatted blocks, and other shared primitives. Together, these guarantees ensure that:

1. switching between light, dark, and neutral variants leaves clinical geometry (grid structure, paddings, radii) invariant;
2. the Synapse-specific skin remains available under a stable `theme.synapse` namespace regardless of the active variant; and
3. focus rings and contrast-sensitive elements always meet conservative accessibility thresholds, an important consideration for clinicians working under time pressure or fatigue.

From a formal perspective, the composed theming stack can be described as a function

$$\Lambda : \mathcal{N} \times \mathcal{D} \times \Sigma \to \tilde{\mathcal{T}},$$

where Σ is the space of Synapse theme configurations, $\tilde{\mathcal{T}}$ the space of merged theme objects, and $\Lambda(\tau, d, \sigma) = \mathrm{merge}(\Theta(\tau, d), \sigma)$ yields the effective theme seen by components. In practice, d and σ are fixed for a given build of the workbench, while τ is clinician-controlled.

Layer 1 — Design tokens
`DESIGN_TOKENS` in `src/constants/design.ts`: spacing scale, color ramps, radii, blur and shadow presets, typography, and animation timings used across the psychiatry workbench.

Layer 2 — Theme variants
`src/styles/theme.ts`: maps active theme names (`light`, `dark`, `neutral`) to concrete `Theme` objects, combining tokens into colors, surfaces, and glassmorphism backgrounds.

Layer 3 — `ThemeContext` provider
`src/contexts/ThemeContext.tsx`: holds (θ_t, τ_t), exposes `setTheme`, `toggleTheme`, and `applyThemeTransition`, and wraps the app in a `StyledThemeProvider` so that all components see a consistent, variant-aware `theme`.

Layer 4 — `AppThemeProvider` and Synapse skin
`src/app/AppThemeProvider.tsx` and `src/theme/GlobalSynapseStyles.ts`: merge the outer theme with `synapseTheme`, inject CSS variables (`-syn-bg-900`, `-syn-gold-300`, ...), and provide a reusable `focusRing` helper for high-contrast keyboard focus.

Figure 27: *Vertical layering of theming responsibilities in SynapseCore, from low-level design tokens to the Synapse clinical skin.*

10.2.3 Clinical and ergonomic considerations

For a psychiatrist, theming is not merely aesthetic. Lighting, contrast, and visual density influence how quickly risk cues are perceived on screen, how fatiguing long documentation sessions become, and whether patients feel overwhelmed by the interface. The `ThemeContext` and `AppThemeProvider` are therefore designed under three clinical constraints:

- *No semantic drift under theme changes.* Risk indicators, scale summaries, and flows must occupy the same spatial locations and preserve visual salience across themes. By keeping geometry and typography controlled by shared tokens and only swapping color ramps, the system maintains a stable "mental map" for the clinician.
- *Low-friction switching.* The `ThemeToggle` button is always available in the header, and the transition function toggle keeps the choice space small. A clinician can adjust the environment between back-to-back sessions (for example, dimming the UI for a distressed patient) without navigating complex settings dialogs.
- *Robust focus and contrast.* The Synapse focus ring and global color variables are tuned so that keyboard navigation, buttons, and high-risk actions (such as marking suicidality) remain clearly visible in all themes. This is particularly relevant for fatigued clinicians and those with mild visual impairments.

In sum, the combined theming stack provides a technically precise yet clinically sensitive foundation on which the rest of the user interface—flows, timer, AI panels, and IDE tools—can safely rely. Subsequent subsections elaborate on the Synapse theme tokens and layout shells that build on this foundation.

10.3 Synapse Theme Tokens

The Synapse theme layer provides a compact but expressive set of design tokens that anchor the entire user interface of the workbench in a consistent, clinically focused visual language. Rather than styling each component in isolation, the theme is defined as a small, typed vocabulary of colours, radii, spacing units, shadows, and focus rings that is shared

across the psychiatry modal, flows, timer, tools (Consulton), and the IDE. This vocabulary is implemented jointly in `src/theme/synapse.ts`, `src/ui/theme/synapseTheme.ts`, `src/ui/theme/semanticTokens.ts`, `src/ui/theme/typography.ts`, and the global CSS baseline in `src/theme/GlobalSynapseStyles.ts`.

At a high level, the theme token system can be viewed as a hierarchy of three layers:

1. *Foundational CSS variables* defined in `GlobalSynapseStyles.ts`, which fix the dark background, accent palette, status colours, radii, spacing units, and base fonts as CSS custom properties (e.g. `-syn-bg-900`, `-syn-gold-500`, `-radius-md`, `-space-3`).
2. *Typed Synapse theme objects* in `synapse.ts` and `synapseTheme.ts`, which lift these variables into TypeScript constants such as `SYNAPSE_COLORS`, `SYNAPSE_ELEVATION`, `SYNAPSE_ACCENT`, `SYNAPSE_LAYOUT`, and the `synapseTheme` instance injected into the styled-components theme.
3. *Semantic tokens* in `semanticTokens.ts` and typography helpers in `typography.ts`, which rephrase the raw palette and geometry into semantically meaningful slots for surfaces, text, borders, status indicators, and focus states.

In combination, these layers ensure that the digital psychiatry workbench presents as a coherent, low-friction environment: a dark, coder-like monospaced surface that reduces visual noise during sessions, while still encoding clinical salience (risk, errors, success) through carefully controlled accent and status colours.

10.3.1 Global CSS token layer

The foundational layer is defined in `src/theme/GlobalSynapseStyles.ts`, which uses `createGlobalStyle` to inject a set of CSS custom properties on `:root`. These properties include:

- **Background and surfaces**: variables such as `-syn-bg-900` and `-syn-surface-800` define the dark base and elevated surfaces used for panels, modals, and cards.
- **Accent and text colours**: `-syn-gold-500`, `-syn-gold-300`, `-syn-text-100`, `-syn-text-400`, and status channels `-syn-danger-400`, `-syn-success-400`, `-syn-warning-400` provide the primary accent (a digital cyan), high-contrast text, and clinical status signalling.
- **Geometry and spacing**: `-radius-sm`, `-radius-md`, `-radius-lg`, `-radius-pill`, as well as `-space-1`–`-space-8` and `-space-x4`–`-space-x24`, define the curvature and spacing rhythm used in almost all components.
- **Typography**: `-font-mono` and `-font-sans` anchor a Coder-style monospaced baseline, with the body element defaulting to `var(-font-mono)` in order to preserve a consistent, IDE-like reading experience even in clinically dense views.
- **AI and auxiliary channels**: a small AI-specific palette (e.g. `-ai-bg`, `-ai-border`, `-ai-gold`) allows the AI assistant panel to be visually differentiated from the rest of the chrome without breaking the overall Synapse aesthetic.

These properties are then re-exported through convenience aliases such as `-color-bg-primary`, `-color-text-primary`, `-color-status-success`, and `-color-accent-primary`, which provide a more semantic naming layer suitable for pure CSS modules. By design, the mapping from semantic names to underlying `-syn-*` variables is one-to-one, so that the palette can be audited centrally without scanning all component styles.

Table 45 summarises the main groups of CSS variables.

Beyond colours and geometry, `GlobalSynapseStyles.ts` also standardises pointer affordances and accessibility features: links inherit the accent colour, `:focus-visible` is rendered as a two-pixel cyan ring around interactive elements, and scrollbars are dark, slim, and fully

Table 45: *Key Synapse CSS custom property groups defined in `GlobalSynapseStyles`. The list is illustrative rather than exhaustive.*

Group	Example variables	Primary role
Background and surfaces	`-syn-bg-900`, `-syn-surface-800`	Dark canvas for the workbench and raised surfaces for panels and modals.
Accent and text	`-syn-gold-500`, `-syn-gold-300`, `-syn-text-100`, `-syn-text-400`	Accent colour for actions and hyperlinks; high-contrast text for low-light clinical environments.
Status channels	`-syn-danger-400`, `-syn-success-400`, `-syn-warning-400`	Encodes error, success, and warning states in flows, timers, and MBC views.
Geometry and spacing	`-radius-md`, `-radius-pill`, `-space-3`, `-space-x16`	Establishes the rounded, glass-like appearance and consistent spacing grid across components.
Typography and AI-specific	`-font-mono`, `-font-sans`, `-ai-bg`, `-ai-gold`	Coder-style monospaced baseline plus subtle differentiation of AI panels while preserving the same palette.

rounded. Collectively, these details reduce cognitive load during sessions and provide clear but unobtrusive cues for keyboard navigation, which is crucial when clinicians interact with the system while simultaneously tracking the patient and their own clinical reasoning.

10.3.2 Typed Synapse theme in `src/theme/synapse.ts`

The second layer, implemented in `src/theme/synapse.ts`, lifts the CSS variables into a typed `SynapseTheme` object compatible with `styled-components`. The type refines the generic `DefaultTheme` into a small, explicit set of design channels:

```
export type SynapseTheme = DefaultTheme & {
  colors: {
    bg900: string; surface800: string; border700: string;
    gold500: string; gold300: string;
    text100: string; text400: string;
    danger400: string; success400: string;
  };
  radius: { sm: string; md: string; lg: string; pill: string };
  space: { x4: string; x8: string; x12: string;
           x16: string; x20: string; x24: string };
  shadow: { elev: string };
  z: { modal: number; popover: number; header: number };
  fonts: { mono: string; sans: string };
  focusRing: (offsetPx?: number) => ReturnType<typeof css>;
};
```

The concrete instance `synapseTheme` then binds these fields to the CSS variables defined by `GlobalSynapseStyles`: `colors.bg900` becomes `var(-syn-bg-900)`, `colors.gold500` becomes `var(-syn-gold-500)`, and so on. In effect, the mapping

$$\theta_{\text{syn}} : K_{\text{css}} \to K_{\text{ts}}$$

takes each CSS custom property `-syn-*` and lifts it into a typed theme key in the TypeScript domain. This lifting not only enables compile-time checking of theme usage, but also forces new components to use a constrained vocabulary instead of ad-hoc colour literals.

The `focusRing` helper provides a canonical focus treatment for interactive components:

$$\texttt{focusRing}(d) = \texttt{outline: } d\texttt{px solid var(-syn-gold-300);}$$

with a default offset of $d = 2$. This helper is surfaced into the application-wide theme via `src/app/AppThemeProvider.tsx`, where the `ThemeProvider` wraps the React tree and exposes `theme.synapse` and `theme.focusRing` to any styled-component. This allows components in clinically salient areas (e.g. risk banners, timer controls, flow buttons) to obtain a consistent, high-visibility focus ring simply by invoking `${props.theme.focusRing()}`.

10.3.3 Synapse palette, elevation, and overlays

While `src/theme/synapse.ts` defines the minimal theme object used by `styled-components`, `src/ui/theme/synapseTheme.ts` collects a richer set of palette and elevation information into constants such as `SYNAPSE_COLORS`, `SYNAPSE_ELEVATION`, `SYNAPSE_ACCENT`, `SYNAPSE_LAYOUT`, `SYNAPSE_ANIM`, and `SYNAPSE_OVERLAY`. Conceptually:

- **`SYNAPSE_COLORS`** defines the canonical dark palette: `bgDark`, `bgSecondary`, `bgTertiary`, `textPrimary`, `textSecondary`, `textAccent`, `goldPrimary`, `goldSecondary`, `blueGray`, and status colours (`success`, `warning`, `error`).
- **`SYNAPSE_ELEVATION`** encodes glassmorphic surfaces and shadows such as `surface`, `surfaceHover`, `surfaceActive`, and multi-layer shadows (`shadowSm`, `shadowMd`, `shadowLg`). These are used to give raised elements (modals, timers, IDE panes) a subtle floating quality while keeping the overall interface dark and low-glare.
- **`SYNAPSE_ACCENT`** and **`SYNAPSE_FOCUS`** define the cyan accent system, including hovered and active states, muted overlays, focus ring width, and offset colour.
- **`SYNAPSE_LAYOUT`** and **`SYNAPSE_ANIM`** provide a compact description of radii, gaps, paddings, and motion curves (fast, base, slow) used consistently in micro-interactions (button presses, tab changes, segmented controls).
- **`SYNAPSE_OVERLAY`** defines the dark, slightly blurred backdrop with a soft vignette used behind modals and the AI assistant, ensuring that clinical content remains foregrounded without harsh contrasts.

These constants are not directly exposed as styled-component theme properties; instead, they underpin semantic tokens (see below) and are consumed by specialised themers such as the Monaco editor theme in `src/components/editor/monacoTheme.ts`. There, the editor background, gutter, selection, and diagnostics colours are all derived from `SYNAPSE_COLORS`, ensuring that the IDE view seamlessly blends into the rest of the workbench and does not feel like a separate application.

10.3.4 Semantic tokens and focus variants

In `src/ui/theme/semanticTokens.ts`, the palette and layout constants are reassembled into a *semantic* token object:

$$\text{SEMANTIC\_TOKENS} = (\text{color}, \text{radius}, \text{space}, \text{shadow}, \text{focus}, \text{aiPalette}).$$

Here, `RADIUS_SCALE` and `SPACING_SCALE` mirror the global `-radius-*` and `-space-*` values, while `SEMANTIC_COLORS` groups tokens into roles such as `surface.primary`, `surface.raised`, `text.muted`, `text.inverse`, `border.subtle`, `border.strong`, `accent.primary`, `accent.subtle`, and status colours. The AI-specific palette (`AI_PALETTE`) isolates colours for

the AI panel and tokens pipeline, preserving the core palette but allowing subtle visual distinction in contexts where the clinician must quickly distinguish machine-generated content from their own notes.

Focus behaviour is centralised through `FOCUS_VARIANTS`, which offers three canonical choices:

- **`default`**: a two-pixel cyan outline derived from `SYNAPSE_COLORS.goldPrimary`, used for most interactive controls.
- **`critical`**: a two-pixel red outline, reserved for destructive or high-risk actions (e.g. deleting a patient, discarding a note).
- **`subtle`**: a one-pixel semi-transparent outline suitable for denser layouts where visual noise must be minimised.

The helper

$$\texttt{focusStyle} : \{\texttt{default}, \texttt{critical}, \texttt{subtle}\} \to \text{CSS}$$

returns the appropriate `outline` and `outline-offset` style string, enabling components to select a focus behaviour without hard-coding colours or line widths. This is particularly important for keyboard-driven workflows in the timer, flows, and IDE panes, where focus must be visible but not overwhelming.

10.3.5 Typography scale and monospaced bias

The typography layer, defined in `src/ui/theme/typography.ts`, enforces a compact Coder-style type scale with a deliberate monospaced bias. The `TYPE_SCALE` object maps symbolic keys (`xs`, `sm`, `md`, `base`, `lg`, `xl`) to pixel sizes, and `LINE_HEIGHTS` fixes corresponding line heights for body text, captions, and code. Helper functions such as `bodyText`, `caption`, and `codeInline` generate small CSS snippets that set `font-family` to `var(-font-code)` (aliasing the monospaced stack), apply the appropriate size, and respect the predefined line height.

An optional responsiveness layer is provided via `enableResponsiveTypography` and `maybeClamp`, which allow font sizes to be expressed as CSS `clamp` expressions. Formally, for a base pixel size p, the helper constructs

$$\text{clamp}(p) = \texttt{clamp}(0.97p,\ p + \delta,\ 1.03p)$$

for a small δ tuned to the viewport. This keeps the typography stable across a variety of screens (desktop, large tablet) while avoiding dramatic fluid-type effects that might be distracting in clinical settings.

Combined with the global body rule `font-family: var(-font-mono)`, this yields a consistent, console-like reading environment that resonates with the IDE metaphor of the application. Clinically, the monospaced bias offers a subtle but practical advantage: aligned digits and uniform character widths facilitate scanning of numerical values (scores, times, dosages) and reduce misreading risks during multitasking.

10.3.6 Vertical stack of theme layers

Figure 28 summarises the vertical stack of theme layers discussed above. The diagram emphasises that the Synapse theme is not a monolithic "skin" but a composition of layered, typed token sets that serve different concerns: foundational palette and geometry, semantic roles, and component-level consumption.

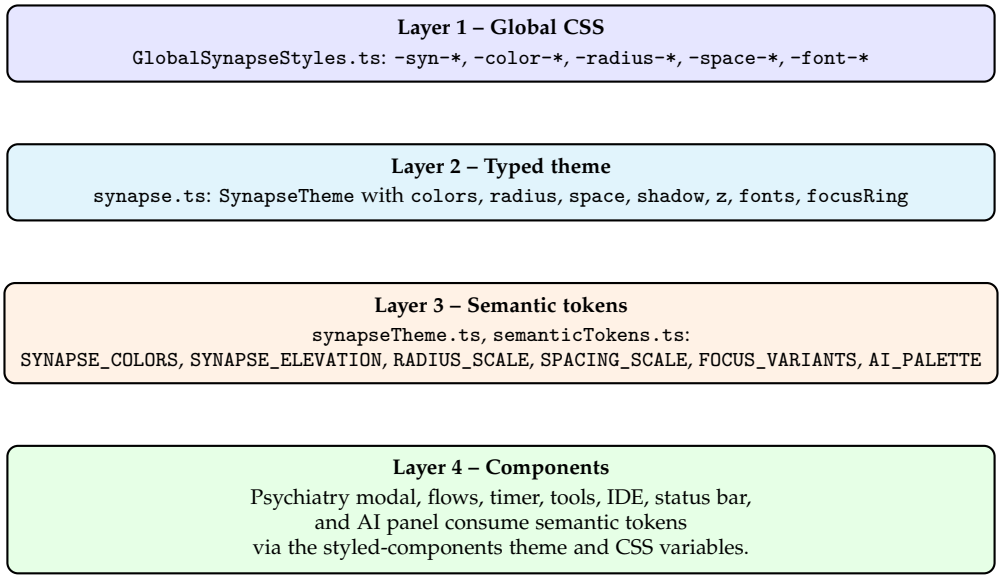


Figure 28: *Vertical stack of Synapse theme layers. Foundational CSS variables are lifted into a typed* `SynapseTheme`*, refined into semantic tokens, and finally consumed by clinical and IDE components.*

From a psychiatric informatics perspective, this layered theme system serves more than aesthetic goals. By constraining the palette, typography, and focus behaviour to a small, centrally controlled set of tokens, the interface can evolve visually (for example, to adopt higher contrast or alternative accent colours) without breaking clinicians' spatial memory of controls. At the same time, the shared tokens enable the AI assistant, timer, flows, and IDE panes to feel like facets of a single clinical workspace rather than disconnected tools, which is essential for safe and efficient use in day-to-day psychiatric practice.

10.4 Center Panel Shell and Clinical Snapshot

The center panel is the primary clinical workspace of SynapseCore. While the application shell (Section 10) establishes a global layout for the entire window, `src/centerpanel/CenterPanelShell.tsx` defines the internal shell that orchestrates the three central columns (left rail, main content, optional right dock), the animated `TopHeader`, the embedded timer controls, and the note-level `ClinicalSnapshotStrip`. This shell is responsible for integrating registry views, structured flows, the note editor, and the tools workspace into a single consistent grid, while preserving accessibility and keyboard navigation across tabs.

10.4.1 Layout shell and tab model

The core component is declared as:

```
// src/centerpanel/CenterPanelShell.tsx
export interface CenterPanelShellProps {
  footerLeft?: React.ReactNode;
  footerRight?: React.ReactNode;
}

type Tab =
  | "Registry"
```

```
  | "New Patient"
  | "Guide"
  | "Note"
  | "Flows"
  | "Tools";

const TABS: Tab[] = [
  "Registry", "New Patient", "Guide", "Note", "Flows", "Tools"
];
```

and the rendered structure begins with a `TimerProvider` and a `section` element using the `.shell` class from `src/centerpanel/styles/centerpanel.module.css`:

```
return (
  <TimerProvider>
    <section
      className={styles.shell}
      data-access-mode={mode}
      aria-label="Center panel area"
      data-reduce-motion={reduceMotion ? true : undefined}
    >
      <a href={`#${MAIN_SCROLL_ROOT_ID}`} className={a11y.skipLink}>
        Skip to main content
      </a>
      <RegistryProvider>
        <TopHeader ...>{/* header content */}</TopHeader>
        <SessionPersistence />
        <div className={`${styles.body} ${styles.bodyNoRightDock}`}>
          {/* tab-specific layout */}
        </div>
        {/* optional footer */}
      </RegistryProvider>
    </section>
  </TimerProvider>
);
```

At the CSS level, the shell is a grid with three vertical bands:

```
/* src/centerpanel/styles/centerpanel.module.css */
.shell {
  height: 100%;
  min-height: 100%;
  display: grid;
  grid-template-rows: auto 1fr auto;
  background: var(--bg);
  color: var(--text);
  font-family: var(--font-mono);
}

.body {
  display: grid;
  grid-template-columns: var(--railW, 240px) minmax(0, 1fr) 360px;
```

```
  gap: var(--gap-3);
  padding: var(--gap-3);
  overflow: hidden;
}

.bodyNoRightDock {
  grid-template-columns: var(--railW, 240px) minmax(0, 1fr);
}
```

so that the center panel is formally decomposed into three regions:

$$\mathcal{R}_{\text{center}} = \{\text{header},\ \text{body},\ \text{footer}\}$$

with the body itself decomposed into

$$\mathcal{C}_{\text{body}} = \{\text{left rail},\ \text{main},\ \text{right dock}\}.$$

Each tab instantiates these regions differently. If we denote the finite tab set by

$$\mathcal{T} = \{\text{Registry},\ \text{NewPatient},\ \text{Guide},\ \text{Note}, \text{Flows},\ \text{Tools}\},$$

then the shell exposes a tab state

$$t \in \mathcal{T}$$

and a layout function

$$\text{layout} : \mathcal{T} \longrightarrow \mathcal{C}_{\text{body}} \to \text{Component}, \tag{10.8}$$

which maps each tab t to a triple of React components for the left rail, main panel, and (optional) right dock.

The following table summarises the main tab configurations as implemented in `CenterPanelShell.tsx`.

Keyboard navigation between tabs is governed by a small finite-state machine inside `TopHeader.tsx`. If we denote the current tab by $t \in \mathcal{T}$ and the last tab elsewhere in the array by $t_{\max}$, and if $e \in \{\text{ArrowLeft}, \text{ArrowRight}, \text{Home}, \text{End}\}$ is a key event, then the transition function

$$\delta_{\text{tab}} : \mathcal{T} \times \{\text{ArrowLeft}, \text{ArrowRight}, \text{Home}, \text{End}\} \longrightarrow \mathcal{T}$$

is implemented in the header as:

$$\delta_{\text{tab}}(t, e) = \begin{cases} \min \mathcal{T}, & e = \text{Home}, \\ \max \mathcal{T}, & e = \text{End}, \\ \text{next}(t), & e = \text{ArrowRight}, \\ \text{prev}(t), & e = \text{ArrowLeft}, \end{cases} \tag{10.9}$$

where next and prev wrap around in the `TABS` array. This ensures that keyboard users can cycle through tabs without ever leaving the semantic `role="tablist"` region.

10.4.2 Top header, timer integration, and accessibility

The top band of the center panel is managed by `src/centerpanel/components/TopHeader.tsx`. It receives the tab list, the active tab, a callback `onTabChange`, a free-form `sessionLabel` (typically derived from `usePersistMeta`), and a live `nowTs` timestamp updated every second:

Table 46: *Center panel tab layout in `CenterPanelShell.tsx`. Each tab instantiates the left rail, main content, and optional right dock within the shared shell and theme.*

Tab	Left rail	Main content	Right dock
Home	Quick shortcuts (recent patients, timers, flows)	`HomePanel` (overview and entry points)	Not used in the current layout.
Session	Session rail (`SessionRail`) with current patient, active session, and timers	`SessionWorkspace` (notes, timers, MSE)	`SessionRightDock` (contextual cards, AI tools).
Flows	Flow rail (`FlowRail`) showing active structured flows and their progress	`FlowWorkspace` (multi-step shells such as agitation, safety, capacity)	`FlowRightDock` (evidence cards, AI summaries).
Registry	Registry rail (`RegistryRail`) summarising patients and encounters	`RegistryMain` (patient detail, encounter list)	Not used in the current layout; reserved for future ML insights or export previews.
New Patient	New-patient guide rail and status (including `OutlineRail` or `OutlineRailV2`)	`NewPatientPage` (structured intake form)	`DraftSnapshotCard` when a draft patient is active; otherwise empty.
Guide	Outline of psychoeducation or guideline cards (`OutlineRail` / `OutlineRailV2`)	`GuideView` or `GuideViewV2`, depending on feature flags	Not used in the current layout.
Tools	Tools patient list (`ToolsPatientList`)	`ToolsActionPanel` (MBC calculators, Consulton exports, utilities)	Not used.

```
<TopHeader
  tabs={TABS}
  activeTab={activeTab}
  onTabChange={(t) => setActiveTab(t as Tab)}
  sessionLabel={sessionName || "session"}
  nowTs={nowTs}
>
  <HeaderTimerButton onOpen={openTimer} />
</TopHeader>
```

Within the header, the `HeaderTimerButton` reads the timer context via `useTimer()` and renders a `TimerButton` whose visual state reflects whether the timer engine is currently running. The actual timer state machine and modal surface are provided by `TimerProvider` and `TimerModal` respectively, so the center panel shell only coordinates entry points (open, close, toggle) via `useTimerModalStore`.

The first child of the shell is an accessibility-oriented skip link:

```
<a href={`#${MAIN_SCROLL_ROOT_ID}`} className={a11y.skipLink}>
  Skip to main content
</a>
```

which allows screen readers and keyboard users to jump directly to the `main` region. The

body is wrapped in ARIA landmarks:

```
<nav aria-label="Left navigation" data-testid="cp-outline">
  {/* left rail */}
</nav>
<main id={MAIN_SCROLL_ROOT_ID} className={styles.main} role="main">
  <section
    key={activeTab}
    className={styles.panelEnter}
    id={`panel-${slug}`}
    role="tabpanel"
    aria-labelledby={`tab-${slug}`}
  >
    {/* main content */}
  </section>
</main>
```

so that the entire center panel behaves like a structured ARIA tabbed interface, consistent across registry, note, flows, and tools.

10.4.3 ClinicalSnapshotStrip: structure and semantics

The clinical snapshot strip is implemented in `src/centerpanel/ClinicalSnapshotStrip.tsx` with styling from `src/centerpanel/clinical-snapshot-strip.module.css`. It is invoked from the Note tab by passing it a set of key–value "pill" descriptors grouped by mode:

```
// src/centerpanel/tabs/Note.tsx
<ClinicalSnapshotStrip
  infoMode={infoMode}
  onSelectInfoMode={setInfoMode}
  overviewPills={pillsByMode.overviewPills}
  riskPills={pillsByMode.riskPills}
  medsPills={pillsByMode.medsPills}
  vitalsPills={pillsByMode.vitalsPills}
  safetyPills={pillsByMode.safetyPills}
  stickyEnabled={false}
  tone="in-header"
/>
```

The strip itself defines a minimal type hierarchy:

```
// src/centerpanel/ClinicalSnapshotStrip.tsx
export type Severity = "ok" | "med" | "high" | "info";

export interface KvPill {
  id: string;
  label: string;
  value?: string;
  unit?: string;
  tooltip?: string;
  severity?: Severity;
```

```
  onClick?: () => void;
  href?: string;
}

export interface ClinicalSnapshotStripProps {
  infoMode: "overview" | "risk" | "meds" | "vitals" | "safety";
  onSelectInfoMode(mode: ClinicalSnapshotStripProps["infoMode"]): void;
  overviewPills: KvPill[];
  riskPills: KvPill[];
  medsPills: KvPill[];
  vitalsPills: KvPill[];
  safetyPills: KvPill[];
  stickyEnabled?: boolean;
  tone?: "note" | "in-header";
  actionRow?: React.ReactNode;
}
```

and a mode-labelled tab row above the key–value pills:

```
const MODE_LABEL: Record<ClinicalSnapshotStripProps["infoMode"], string> = {
  overview: "Overview",
  risk:     "Risk",
  meds:     "Meds",
  vitals:   "Vitals",
  safety:   "Safety",
};
```

Formally, let $\mathcal{M}$ be the set of snapshot modes,

$$\mathcal{M} = \{\text{overview},\ \text{risk},\ \text{meds},\ \text{vitals}, \text{safety}\},$$

and let $\mathcal{C}$ denote the space of de-identified clinical states for the current patient and encounter (including scale scores, risk grades, psychotropic regimen, and recent vital signs). A single pill instance can be modelled as

$$p = (\text{id}, \ell, v, u, \sigma) \in \mathcal{P},$$

where ℓ is a label (e.g. “PHQ-9”), v is a value (e.g. a numeric score), u is an optional unit, and $\sigma \in \{\text{ok}, \text{med}, \text{high}, \text{info}\}$ is a severity class. The snapshot strip is then a view

$$\mathsf{snapshot} : \mathcal{M} \times \mathcal{C} \longrightarrow \mathcal{P}^*, \tag{10.10}$$

mapping a mode $m \in \mathcal{M}$ and a clinical state $c \in \mathcal{C}$ to a finite sequence of pills $p_1, \ldots, p_k$.

Internally, each mode is resolved via a simple switch:

$$\mathsf{snapshot}(m, c) = \begin{cases} \mathsf{buildOverview}(c), & m = \text{overview}, \\ \mathsf{buildRisk}(c), & m = \text{risk}, \\ \mathsf{buildMeds}(c), & m = \text{meds}, \\ \mathsf{buildVitals}(c), & m = \text{vitals}, \\ \mathsf{buildSafety}(c), & m = \text{safety}, \end{cases} \tag{10.11}$$

where each builder function extracts and formats a different subset of the clinical state: recent measurement-based care scores and grades for buildOverview, self-harm and violence risk flags for buildRisk, current psychotropic regimen for buildMeds, basic physiological parameters for buildVitals, and contractual or environmental safety constraints for buildSafety.

Severity decoration is delegated to a helper $\sigma : \mathcal{C} \to \{\text{ok}, \text{med}, \text{high}, \text{info}\}$ that maps derived risk variables to visual emphasis. For example, if $G \in \{1,2,3,4,5\}$ is a discrete risk grade from the measurement module, a typical mapping is:

$$\sigma(G) = \begin{cases} \text{ok}, & G \in \{1,2\}, \\ \text{med}, & G = 3, \\ \text{high}, & G \in \{4,5\}. \end{cases} \tag{10.12}$$

The resulting severity is encoded into the DOM via a `data-severity` attribute:

```
<span className={styles.kvPill} data-severity={sev} ...>
  {/* label + value */}
</span>
```

and interpreted by the CSS module:

```
/* src/centerpanel/clinical-snapshot-strip.module.css */
.kvPill[data-severity="med"] {
  background-color: var(--risk-med-bg);
  color: var(--risk-med-fg);
  border-color: rgba(255,190,0,0.45);
}
.kvPill[data-severity="high"] {
  background-color: var(--risk-high-bg);
  color: var(--risk-high-fg);
  border-color: rgba(255,60,60,0.55);
}
```

which reuses the risk palette derived from the Synapse theme tokens (Section 10.3).

The strip wrapper itself is sticky when `stickyEnabled` is set:

```
.stripWrap {
  position: sticky;
  top: var(--strip-top, calc(var(--topbarHeight, 0px)
       + var(--patientHeaderHeight, 44px)));
  z-index: 90;
  background-color: var(--surface, #0f141c);
  border-bottom: 1px solid rgba(255,255,255,0.06);
  padding-top: 6px;
  padding-bottom: 6px;
  padding-left: var(--note-h-gutter);
  padding-right: var(--note-h-gutter);
  display: grid;
  row-gap: 6px;
}
```

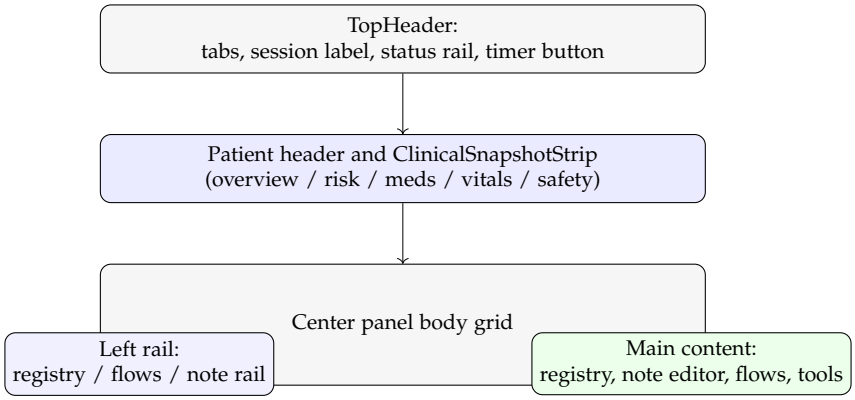


Figure 29: *Vertical layout of the center panel shell. The `TopHeader` hosts navigation, session label, and timer entry; the patient header and `ClinicalSnapshotStrip` provide a continuously visible summary of risk and key metrics in the Note tab; and the body grid hosts the tab-specific left rail and main content.*

so that in long notes the snapshot remains anchored directly beneath the header while the editor scrolls beneath it. This makes the current risk grade, recent scores, medication changes, and safety constraints visually available throughout the editing process, without requiring the clinician to switch tabs or leave the note context.

10.4.4 Keyboard interaction and vertical layout

Both the tab row in `TopHeader` and the mode row in `ClinicalSnapshotStrip` follow the same keyboard pattern: arrow keys move focus left or right, and Enter or Space activates the focused tab. The snapshot component keeps a small ref array of the tab buttons:

```
const tabRefs = React.useRef<Array<HTMLButtonElement | null>>([]);

function onTabsKeyDown(e: React.KeyboardEvent<HTMLDivElement>) {
  if (e.key !== "ArrowLeft" && e.key !== "ArrowRight") return;
  e.preventDefault();
  const modes = ["overview","risk","meds","vitals","safety"] as const;
  const curIdx = modes.indexOf(infoMode);
  let nextIdx = curIdx;
  if (e.key === "ArrowLeft")
    nextIdx = (curIdx - 1 + modes.length) % modes.length;
  if (e.key === "ArrowRight")
    nextIdx = (curIdx + 1) % modes.length;
  const next = modes[nextIdx];
  const btn = tabRefs.current[nextIdx];
  if (btn) { try { btn.focus(); } catch {} }
  onSelectInfoMode(next);
}
```

which can be seen as a mode-specific analogue of the global tab FSM in (10.9).

The overall vertical composition of the center panel shell is shown in Figure 29. From top to bottom, the clinician sees the animated header with tabs and timer entry point, the patient header with snapshot strip (in the Note tab), and the main two-column body grid.

Through this combination of a typed tab model, a grid-based layout shell, and a compact but expressive clinical snapshot strip, the center panel supports fast alternation between reg-

istry, flows, note editing, and tools while keeping the most safety-critical information (risk and recent scores) anchored in the clinician's visual field.

10.5 Atoms and Shared Components

The SynapseCore interface is built on an explicit atomic design philosophy. Instead of styling each screen ad hoc, the system defines a small set of *atoms*—generic, theme-aware React components such as buttons, inputs, neural background canvases, and glassmorphism cards—that can be safely reused across the clinical flows, timer, tools, and IDE modules described in Section 10. These atoms are implemented in `src/components/atoms/*` (e.g. `Button.tsx`, `Input.tsx`, `NeuralBackground.tsx`, `NeuralGlassCardFinal.tsx`, `StatusBar.tsx`, `Logo.tsx`) and are then assembled into shared templates such as the Synapse landing page and its hero section.

From a formal perspective, each atomic component can be regarded as a typed rendering function

$$\mathcal{A}_k : \mathcal{P}_k \times \Theta \times \Sigma \longrightarrow \mathcal{V},$$

where $\mathcal{P}_k$ is the props space for atom k (e.g. `ButtonProps`, `InputProps`), Θ is the current Synapse theme (Section 10.3), Σ is the interaction state space (e.g. idle, hover, active, focused, disabled), and $\mathcal{V}$ is the React virtual DOM. Composite components and templates then become compositions of such atoms, i.e.

$$\mathcal{T} = \phi(\mathcal{A}_1(p_1, \theta, \sigma_1), \ldots, \mathcal{A}_n(p_n, \theta, \sigma_n)),$$

where ϕ is a pure layout function. This separation of concerns ensures that visual and behavioural consistency is maintained even as new clinical flows or AI features are added.

Table 47 summarises the main atoms and their clinical roles.

10.5.1 Atomic model and composition

Within the implementation, each atom is written as a small, theme-aware functional component. For the button atom, for example, the core export takes the form

$$\text{Button} : \mathcal{P}_{\text{btn}} \times \Theta \times \Sigma_{\text{btn}} \to \mathcal{V}, \tag{10.13}$$

where Σ_{btn} includes states such as `default`, `hover`, `active`, `focusVisible`, and `disabled`. The visual style is determined by:

$$\text{style}_{\text{btn}} = g\big(\texttt{variant}, \texttt{size}, \theta, \sigma\big),$$

with `variant` and `size` taken from the props and (θ, σ) derived from the current theme and user interaction. The functions `ButtonVariants` and `ButtonSizes` in `Button.tsx` implement g as a composition of `styled-components` mixins.

More generally, any composite UI region—for instance, the psychiatry modal header or an AI suggestion tray—can be written as a pure composition

$$\mathcal{C} = \psi\big(\text{Button}(\cdot), \text{Input}(\cdot), \text{NeuralGlassCard}(\cdot), \ldots\big),$$

where ψ is a stateless layout function. This formalisation allows the same atoms to be reused in different clinical contexts (e.g. risk flows, treatment planners, AI consoles) without duplicating styling logic or diverging from the theme tokens.

Figure 30 shows a vertical schema of how the Synapse theme feeds into atoms, shared components, and finally homepage-level templates.

Table 47: Overview of core atomic components in `src/components/atoms`. Paths are indicative and shortened for readability.

Atom	Primary type	Key files	Clinical / UX role
`Button`	Action control	`components/atoms/Button.tsx`	Executes confirm, cancel, navigation, and AI actions with variant-specific risk signalling (primary, secondary, danger, segmented, icon-only).
`Input`	Data field	`components/atoms/Input.tsx`	Handles typed user input (e.g. names, IDs, prompts) with floating labels, error highlighting, and character-count feedback.
`NeuralBackground`	Ambient canvas	`components/atoms/NeuralBackground.tsx`	Renders a dynamic neural field that visually encodes the theme and provides an ambient cognitive metaphor for the workbench.
`NeuralGlassCard`	Glassmorphism card	`components/atoms/NeuralGlassCardFinal.tsx`	Presents key concepts (e.g. multi-model orchestration, psychiatry co-pilot) as animated hero cards on the Synapse landing layout.
`StatusBar`	Global status strip	`components/atoms/StatusBar.tsx`	Displays environment, connectivity, time, and shortcut hints at the bottom of the workspace.
`Logo`, `Icon`, `Toast`	Branding / feedback	`Logo.tsx`, `Icon.tsx`, `Toast.tsx`	Provide consistent branding, iconography, and transient toast notifications across modules.

10.5.2 Button atom: risk-aware action semantics

The button atom in `Button.tsx` encapsulates the majority of the workbench's action semantics. Variants are defined in a `ButtonVariants` object using `css` helpers from `styled-components`. Representative variants include:

- *Primary*: filled, gold-accented actions used for clinically important confirmations (e.g. saving a note, advancing a flow step).
- *Secondary*: neutral glass surface buttons suitable for non-destructive actions and toggles.
- *Ghost* and *subtle ghost*: minimally styled buttons used in dense toolbars or low-salience controls.
- *Danger*: red-accented actions used for destructive or high-risk operations (e.g. clearing a session, deleting a case).
- *Pill* and *icon-only*: compact controls for chip-like filters or icon-only actions in the status bar and AI console.

Each variant first defines a default background, border, and text colour using `SYNAPSE_ACCENT`, `SYNAPSE_ELEVATION` and related tokens, and then refines these for `:hover`, `:active`, and `:disabled` states. For a given variant v and interaction state σ, the effective colour triplet $(c_{\text{bg}}, c_{\text{text}}, c_{\text{border}})$ can be viewed as:

$$(c_{\text{bg}}, c_{\text{text}}, c_{\text{border}}) = h_{\text{btn}}(v, \sigma; \theta),$$

where h_{btn} is entirely determined by theme tokens (e.g. `SYNAPSE_ACCENT.gold`, `SYNAPSE_ELEVATION.surfaceHover`) and state.

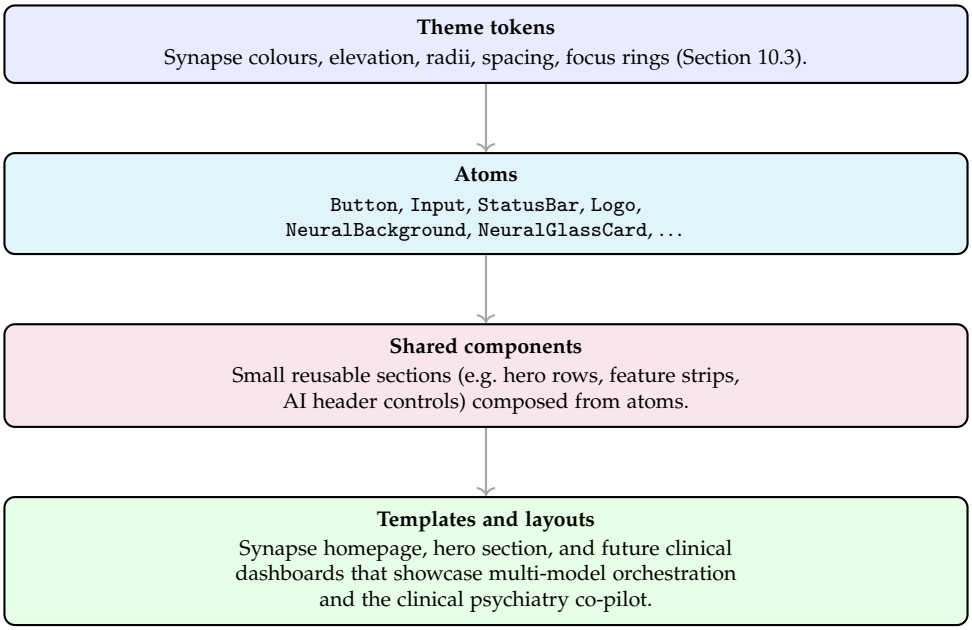


Figure 30: *Atoms and shared components as a vertical stack above the Synapse theme. Theme tokens inform atomic components, which compose into shared sections and finally into templates such as the hero landing layout.*

Button sizes are parameterised in the `ButtonSizes` object, which sets padding, font size, minimum height, and radius. This makes the choice of density (small/medium/large) explicit and helps maintain ensured hit-target sizes across views, which is crucial when the clinician is working under time pressure.

Finally, the button's `:focus-visible` state is delegated to `SYNAPSE_FOCUS`, ensuring a consistent cyan focus ring with appropriate outline offset regardless of variant. This systematic focus handling is essential for keyboard-only navigation and supports accessibility guidelines without duplicating CSS across components.

10.5.3 Input atom: clinical data entry fields

The input atom `Input.tsx` encapsulates a glassmorphism text field with floating label, theme-aware focus ring, and explicit error state. Its internal structure can be summarised as:

- `InputWrapper`: a relatively positioned container used to anchor the floating label, optional description, and character counter.
- `StyledInput`: an HTML `<input>` element styled with Synapse elevation, spacing, and focus tokens.
- `FloatingLabel`: a label that transitions from a mid-line position to the top of the field when focused or when the input contains a value.
- Optional affordances such as clear-button, helper text, and character-count display.

Two boolean props, `hasError` and `hasLabel`, are propagated into the styled input. Formally, setting `hasError` corresponds to switching the border colour from the neutral token to the error status channel:

$$c_{\text{border}} = \begin{cases} c_{\text{neutral}}, & \text{if } \texttt{hasError} = \text{false}, \\ c_{\text{error}}, & \text{if } \texttt{hasError} = \text{true}, \end{cases}$$

where c_{neutral} and c_{error} are derived from `SYNAPSE_ELEVATION.border` and `-color-error`,

respectively. The focus state uses `SYNAPSE_FOCUS.width`, `SYNAPSE_FOCUS.ring`, and `SYNAPSE_FOCUS.ringOffset` to draw a two-layer glow around the field.

The floating label's position is computed from a discrete state $(\ell_{\text{focused}}, \ell_{\text{hasValue}}) \in \{0,1\}^2$:

$$\text{labelPos} = \begin{cases} \text{midline}, & \ell_{\text{focused}} = 0,\ \ell_{\text{hasValue}} = 0, \\ \text{top}, & \text{otherwise}, \end{cases}$$

which is implemented via a CSS conditional on props `isFocused` and `hasValue`. This small state machine ensures that labels never occlude typed content and that the field remains comprehensible even when empty (the label returns to the midline).

Internally, the component also supports both controlled and uncontrolled usage via an internal value state. A helper type `SynapseInputEl` enriches the DOM element with private fields used by Synapse flows, allowing diagnostic or debugging tooling to attach listeners without leaking implementation details into clinical code.

10.5.4 Neural visual atoms: background and glass cards

Beyond purely functional atoms, SynapseCore defines neural visual atoms that carry the cognitive metaphors of the workbench.

Neural background. `NeuralBackground.tsx` renders a large canvas behind the hero section or other high-visibility layouts. Conceptually, it maintains two coupled sets of objects:

- A set of nodes $\{n_i\}_{i=1}^{N}$ with positions (x_i, y_i, z_i), velocities (v_i^x, v_i^y, v_i^z), and attributes such as `type`, `energy`, and `pulsePhase`.
- A set of weighted connections $\{(i, j, w_{ij})\}$ linking nearby nodes into a dynamic network.

Nodes are initialised using either a random spherical distribution or a Fibonacci spiral on the unit sphere. For the spherical layout `generateSphereLayout`, positions are sampled as

$$\phi \sim \mathcal{U}(0, 2\pi), \quad \cos\theta \sim \mathcal{U}(-1, 1), \quad r > 0,$$

and then mapped to Cartesian coordinates:

$$x = r\sin\theta\cos\phi, \quad y = r\sin\theta\sin\phi, \quad z = r\cos\theta.$$

Connections are created between nodes that fall below a distance threshold, with an additional Bernoulli trial (probability 0.33) that determines whether the edge is actually drawn. Each edge is assigned a random strength $w_{ij} \in [0.3, 1.0]$, which influences line opacity.

Theme transitions are handled by a dedicated ref object, `themeTransitionRef`, which stores the previous and target colour sets along with animation timings. When the theme changes from θ_{old} to θ_{new}, a short transition of duration approximately 800 ms interpolates node and edge colours, yielding a smooth morphing of the background rather than an abrupt switch. A `prefers-reduced-motion` media query is honoured to respect user accessibility preferences; when this flag is set, motion is dampened and glitch-like effects are reduced.

Neural glass cards. `NeuralGlassCardFinal.tsx` implements a theme-aware, animated glassmorphism card used to highlight key product concepts such as multi-model orchestration, the clinical psychiatry co-pilot, or observability features. The component combines three ideas:

1. *Glass surface*: a semi-transparent background with theme-dependent colour, soft blur, and glow derived from the theme context.
2. *Glitch animation*: keyframe sequences (`glitchLight`, `glitchDark`, `glitchNeutral`) that briefly perturb the card's border and transform when the theme or card index changes, creating a subtle "activation" effect.
3. *Framer Motion transitions*: a `motion.div` container specifies spring-like entrance and hover animations, with the card gently translating and scaling into place.

The helper `getThemeStyles(themeName)` returns a small map of colours for icon backgrounds, title text, and body text. Formally, we can view this as a lookup

$$s_{\text{card}} : \Theta \to \{c_{\text{icon}}, c_{\text{iconBg}}, c_{\text{title}}, c_{\text{text}}\},$$

which decouples card content from theme choice. As with other atoms, the card respects the monospaced bias of the workbench, using a coder-style font stack for titles and descriptions.

Taken together, the neural background and neural glass cards form an ambient representation of the system's purpose: a programmable, neural-like fabric underlying the more traditional clinical components.

10.5.5 Homepage templates and introduction cards

Although atoms are designed to be usable in any part of the application, they come together most visibly in the Synapse landing page and hero layout. Conceptually, the `SynapseHomepage` template and its `HeroSection` perform the following composition:

1. Lay out a full-screen `NeuralBackground` behind the main content, binding it to the current theme and motion preferences.
2. Display the Synapse logo and navigation controls at the top, using atomic `Logo`, `Button`, and `ThemedDropdown` components.
3. Arrange a small grid or horizontal scroll of `NeuralGlassCard` instances, each card corresponding to a major capability:
   - multi-model orchestration of AI providers;
   - clinical psychiatry co-pilot for assessment and treatment planning;
   - measurement-based care engine and psychometric scoring;
   - observability and audit tooling for AI safety.
4. Route user interactions (e.g. "Open psychiatry workspace", "Explore tools") into the main application shell, where the same atoms are reused in more clinical contexts.

In the written manuscript, these hero cards serve a dual purpose. At runtime they act as an onboarding surface for clinicians and researchers; in print, they provide a visually grounded way to connect the technical contributions of SynapseCore (multi-model orchestration, flows, timer, tools, IDE) with the narrative arc of the paper's Introduction and Contributions sections. Because the cards are built from atoms and theme tokens rather than static artwork, the same visual language carries through to later screenshots and figures, reinforcing that the system being described is a coherent, living workbench rather than a collection of disconnected prototypes.

Table 48: Core panes of the Enhanced IDE and their primary responsibilities.

Pane	Implementation	Primary responsibilities
File explorer	`src/components/file-explorer/FileExplorer.tsx`, `FileExplorerHeader.tsx`	Presents the project tree, supports creation, renaming, moving, and deleting of files/folders with drag–and–drop and keyboard navigation; exposes import/export for project archives and starter templates.
Editor	`src/components/editor/MonacoEditor.tsx`, `monacoTheme.ts`	Renders the active text buffer with a Synapse-specific Monaco theme; propagates edits to the editor store; exposes AI-assisted refactoring actions and preview tooling while feeding telemetry to the status bar.
AI panel	`src/components/ai/panel/SynapseCoreAIPanel.tsx`	Hosts the conversational AI assistant configured in Section 4; displays chat history, tool calls, and apply-plan proposals; is resizable and can be toggled without losing state.
Terminal / log	`src/components/terminal/components/Terminal.tsx`, `terminalLogBus.ts`	Streams textual logs (build output, apply-plan dry runs, task results) from the terminal bus; supports clearing and filtering while preserving a chronological, non-editable record.

11 Enhanced IDE and Developer Tooling

11.1 Enhanced IDE Core

The Enhanced IDE surface is the primary locus where SynapseCore exposes its developer-facing affordances while remaining closely aligned with the clinical shell described in earlier sections. Implemented in `src/components/ide/EnhancedIDE.tsx`, this surface presents a four-pane layout composed of a file explorer, a Monaco-based code editor, an AI panel, and a terminal/log strip. These panes are not merely visual conveniences: they form a tightly coordinated environment that allows a psychiatrist–developer to inspect configuration, edit domain models, apply AI-generated refactorings, and observe system output without leaving the clinical context.

From a systems perspective, the Enhanced IDE can be characterised as a pure rendering function over three high-level state spaces:

$$\mathcal{L} \times \mathcal{E} \times \mathcal{F} \longrightarrow \text{JSX},$$

where $\mathcal{L}$ denotes the layout state (pane sizes, visibility flags), $\mathcal{E}$ the editor state (active tab, dirty buffers, selection), and $\mathcal{F}$ the project file tree. Concretely, these are backed by Zustand stores such as `useAppStore`, `useEditorStore`, and `useFileExplorerStore`, together with derived selectors (`useActiveTab`, `useDirtyTabs`) and imperative actions (`useLayoutActions`, `useTabActions`, `useFileOperations`). The component itself is wrapped in an `IdeThemeScope` to ensure that all descendants inherit the IDE-specific color tokens and typography defined in the Synapse design system.

Table 48 summarises the principal panes and their implementation entry points.

11.1.1 Layout model and pane coordination

The layout state managed by `useAppStore` encodes both geometric and visibility constraints. In simplified form, we can express the layout vector for a given viewport as

$$\ell = \big(W,\ w_{\text{sidebar}},\ w_{\text{ai}},\ h_{\text{ai}},\ c_{\text{sidebar}},\ c_{\text{terminal}}\big),$$

where W is the viewport width, w_{sidebar} the width of the file explorer, w_{ai} the width of the AI panel, h_{ai} its height, and $c_{\text{sidebar}}, c_{\text{terminal}} \in \{0,1\}$ indicate whether the sidebar and terminal are collapsed.

In `EnhancedIDE.tsx`, the horizontal layout of the upper band is governed by three constraints:

1. A minimum and maximum sidebar width $(w_{\min}, w_{\max})$, corresponding to `MIN_SIDEBAR` and `MAX_SIDEBAR`.
2. A minimum editor width $w_{\text{editor,min}}$, corresponding to `MIN_EDITOR_WIDTH`.
3. The current AI panel width w_{ai}, which may be 0 when the panel is hidden.

When the user drags the vertical splitter between the file explorer and the editor, the new proposed width $\widehat{w}_{\text{sidebar}} = w_{\text{sidebar}} + \Delta$ is clamped according to

$$w'_{\text{sidebar}} = \min\Big(\max\big(\widehat{w}_{\text{sidebar}},\ w_{\min}\big),\ \min\big(w_{\max},\ W - w_{\text{ai}} - w_{\text{editor,min}}\big)\Big), \tag{11.1}$$

ensuring that the remaining space for the editor never falls below $w_{\text{editor,min}}$. This logic is implemented imperatively in the mouse-move handler attached to the splitter; however, it is conceptually a projection $\Pi_{\mathcal{C}}(\widehat{w}_{\text{sidebar}})$ onto a convex feasible set $\mathcal{C}$ defined by the above bounds.

An analogous constraint applies vertically to the AI assistant panel, which is anchored to the right edge of the viewport with a maximum height of `calc(100vh - 26px)`, leaving room for the global status bar. The `layout.aiAssistantHeight` value is continuously updated as the user drags the horizontal splitter, with clamping between a minimum height (e.g., ≈ 240 pixels) and the current viewport height minus the fixed status bar.

Taken together, these constraints yield a multi-pane layout that remains stable under resizing and theme changes. The terminal strip occupies the lower edge of the viewport, and its visibility is toggled via `toggleTerminal()` from `useLayoutActions`, together with keyboard accelerators and command-palette entries registered through `src/services/commandRegistry.ts`.

Figure 31 provides a schematic, vertical overview of the Enhanced IDE layout.

11.1.2 File explorer and project tree

The file explorer implemented in `src/components/file-explorer/FileExplorer.tsx` and its Synapse-styled header in `FileExplorerHeader.tsx` together provide a domain-aware view over the project tree. At the type level, a project is represented as a rooted tree $\mathcal{F} = (V, E)$ where each vertex $v \in V$ corresponds to a `FileNode` and each edge $(u, v) \in E$ indicates containment (*folder* u contains child v). Conceptually:

$$\mathcal{F} = \big(V, E, r\big), \qquad r \in V,$$

with a designated root r representing the project root directory.

Each node carries metadata:

$$v = \big(\mathit{id},\ \mathit{name},\ \mathit{path},\ \mathit{kind},\ \mathit{language},\ \mathit{template},\ \ldots\big),$$

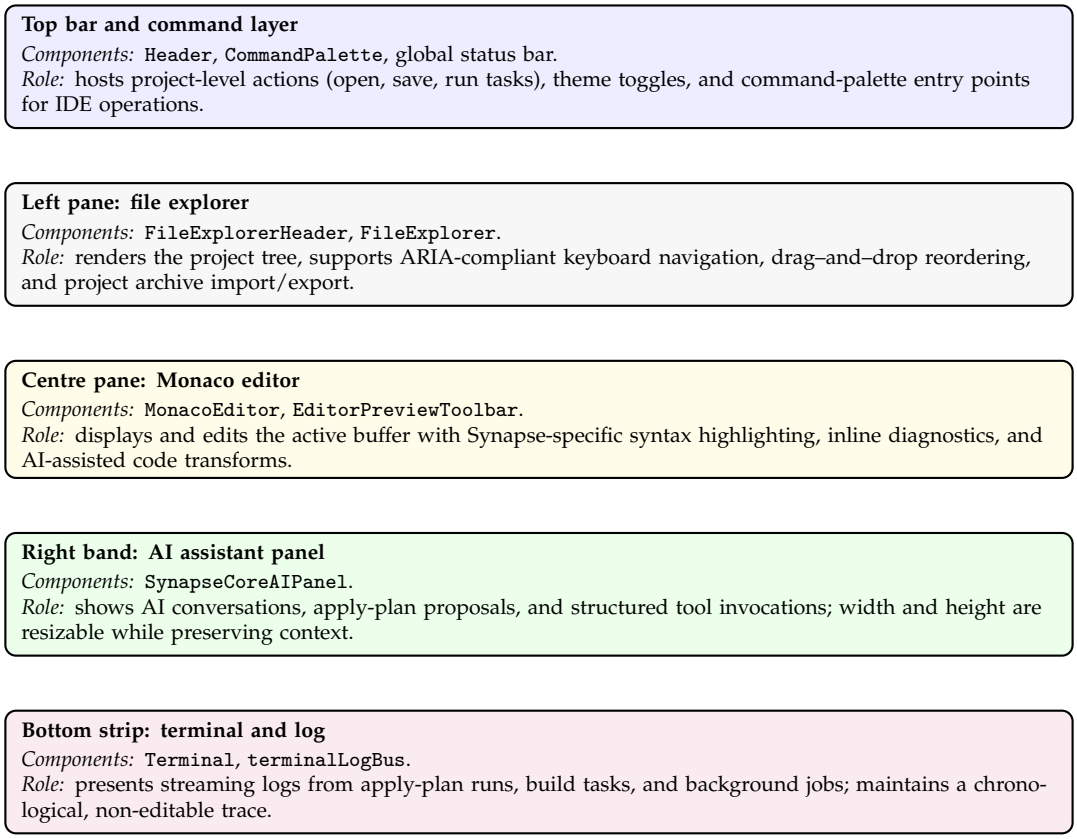


Figure 31: *Vertical schematic of the Enhanced IDE layout. The four panes share a common layout state and theme scope, allowing psychiatric configuration work, AI-assisted refactoring, and observability to coexist within a single workspace.*

where *kind* $\in$ {file, folder} and *language* encodes the logical language (TypeScript, YAML, LaTeX, etc.), not necessarily tied to the file extension. This metadata is later used by higher-level tooling, for example to propose AI refactorings that are specific to prompt YAML files versus TypeScript orchestration code.

The explorer renders this tree as an ARIA-compliant `role="tree"` with `role="treeitem"` for each node and `aria-level`, `aria-expanded`, and `aria-activedescendant` attributes set to enable screen-reader navigation. Keyboard handlers implement a canonical set of bindings:

- *Arrow keys* move the focus up and down the visible tree, expand/collapse folders, and avoid stepping into disabled nodes.
- *Enter* or *Space* activates the selected node (opening a file or toggling a folder).
- *F2* or *Cmd/Ctrl+R* initiates inline rename via the `InlineRename` component.

Drag–and–drop interactions allow users to move files between folders, and hover affordances indicate valid drop targets via CSS classes such as `drop-target-valid` and `drop-target-invalid`. Formally, let $\sigma_{\mathcal{F}}$ denote the current file tree state and $\texttt{MOVE}(u, v)$ the operation of moving node u under destination folder v. The explorer realises a transition function

$$\delta_{\text{tree}} : (\sigma_{\mathcal{F}}, \texttt{MOVE}(u, v)) \longrightarrow \sigma'_{\mathcal{F}},$$

subject to invariants such as acyclicity (a node cannot be moved inside its own subtree) and type constraints (a file cannot be made a parent). Similar transition functions exist for `CREATE_FILE`, `CREATE_FOLDER`, `RENAME`, and `DELETE` operations, which are wired to the `useFileOperations` hook.

The header component `FileExplorerHeader.tsx` wraps these operations in a Synapse-

Table 49: *Illustrative file operations in the Enhanced IDE and their effects on the project tree.*

Operation	UI gesture / command	Effect on tree state $\sigma_{\mathcal{F}}$
Create file	"New file" button in header, or context menu on folder	Inserts a new leaf node u with *kind* = file and the selected language/template as a child of the target folder; opens a corresponding editor tab.
Create folder	"New folder" button or context menu	Inserts a new internal node u with *kind* = folder; updates ARIA levels and sorts within the containing folder according to explorer rules.
Rename	Inline rename (F2) or context-menu "Rename"	Updates *name* and derived *path* for node u, propagating path updates to all descendants to preserve the tree invariant.
Move	Drag–and–drop onto another folder	Removes edge (p, u) from E and inserts (v, u) for destination folder v, after verifying that v is not in the subtree of u; updates ordering and focus.
Delete	"Delete" (keyboard) or trash icon in context menu	Removes node u and all its descendants from V and E; closes any open editor tabs associated with files under u.

styled control strip. It provides:

1. A contextual search box bound to `searchQuery`, which filters the in-memory tree while preserving the original hierarchy.
2. Primary actions for creating files and folders, including language-aware templates (e.g. TypeScript module, YAML config, LaTeX manuscript).
3. Import/export affordances using `JSZip`, enabling the project tree to be packed into or restored from a ZIP archive. This mechanism is particularly relevant for clinical research workflows where an entire configuration (prompts, flows, models) must be archived as a reproducible snapshot.

Table 49 lists the principal file operations exposed by the header and context menu, together with their state transitions.

11.1.3 Monaco editor and Synapse theme integration

The central editing surface inside the Enhanced IDE is the `MonacoEditor` component in `src/components/editor/MonacoEditor.tsx`, which wraps the `@monaco-editor/react` bridge and applies a Synapse-specific theme via `monacoTheme.ts`. The theme definition function `defineSynapseMonacoTheme` is invoked at editor initialisation to register a palette based on `SYNAPSE_COLORS`, including background (`bgDark`, `bgSecondary`), foreground (`textPrimary`, `textSecondary`), and semantic hues for errors, warnings, and informational diagnostics.

At runtime, the active theme identifier is selected as a function of the global UI theme:

$$g(\tau) = \begin{cases} \texttt{'synapse-pro-light'}, & \text{if } \tau = \text{light}, \\ \texttt{'synapse-pro-neutral'}, & \text{if } \tau = \text{neutral}, \\ \texttt{'synapse-pro'}, & \text{otherwise (dark)}. \end{cases}$$

Here τ is obtained from the `ThemeContext` and the resulting string is passed to the Monaco

Table 50: *Mapping between global Synapse themes and Monaco editor theme identifiers.*

Global theme τ	Monaco theme id $g(\tau)$	Visual characteristics
`light`	`'synapse-pro-light'`	High-contrast surfaces with pale background and subtle grid; optimized for print-like readability when authoring manuscripts or clinical reports.
`neutral`	`'synapse-pro-neutral'`	Low-saturation, mid-tone background suitable for prolonged reading of configuration and YAML files without excessive contrast.
`dark`	`'synapse-pro'`	Deep background with gold accent tokens; emphasises syntax coloring and inline diagnostics for intense coding sessions.

`Editor` as the `theme` prop. Table 50 summarises the mapping between global and editor-level themes.

Beyond theming, the editor is tightly integrated with telemetry and status-bar infrastructure. As the user invokes AI-assisted code actions (for example from the editor toolbar), the helper `estimateTokens` computes a rough token-count approximation for the concatenation of the prompt and the current code snippet. Let x denote the natural-language instruction and c the code segment; the approximated token count is

$$\widehat{T} = f_{\text{tok}}(x, c),$$

where f_{tok} is a deterministic approximation based on character length and heuristic token boundaries. This value is passed to `timeAndLog`, which wraps the asynchronous API call exec : $\varnothing \to$ Completion and records both latency and $\widehat{T}$ into the telemetry subsystem described in Section 4. The resulting completion is then parsed either as a unified diff (via `parseUnifiedDiffToPatched`) or as a plain-text replacement, after which the editor state is updated via `useTabActions`.

The editor further subscribes to an `editorBridge` bus via `subscribeEditorBridge`, enabling other parts of the system (AI apply-plan executors, quick-fix buttons, terminal tasks) to request operations such as *insert at cursor*, *replace selection*, or *jump to definition* in a decoupled manner. Conceptually, this bus implements a family of partial actions

$$\delta_{\text{editor}} : (\sigma_{\mathcal{E}},\, a) \longrightarrow \sigma'_{\mathcal{E}},$$

where $\sigma_{\mathcal{E}}$ denotes the editor state (content, cursor, selection, scroll position) and a belongs to a fixed vocabulary (`insertAtCursor`, `replaceRange`, `focus`, etc.). This functional viewpoint is crucial for ensuring that AI tooling can modify buffers without bypassing the ordinary undo stack or violating invariants.

11.1.4 AI panel and terminal as structural neighbours

Although the detailed semantics of the AI panel and its bridges to the editor are elaborated in Section 11.2, it is important at the core layout level to note that both the `SynapseCoreAIPanel` and the `Terminal` are treated as first-class structural neighbours of the editor, not as external overlays. In `EnhancedIDE.tsx`, the AI panel is positioned as a fixed element on the right edge, with its width driven by `layout.aiAssistantWidth` and mirrored in CSS transitions so

that show/hide operations remain visually smooth. The panel interacts with the same theme scope and typography tokens as the editor, emphasising that it is a co-equal part of the IDE rather than a separate chat widget.

Similarly, the terminal is pinned to the bottom of the viewport, with height controlled by layout state and log content streamed via the `terminalLogBus`. When an apply-plan dry run or task execution is triggered (e.g. via `triggerTask` from `services/tasksBridge.ts`), structured messages are rendered line by line with timestamps and severity indicators, giving the psychiatrist–developer immediate feedback on configuration changes and AI-assisted refactorings.

Together, these structures realise the central design goal of the Enhanced IDE: a unified, instrumented workspace in which domain configuration, AI interaction, and system observability are co-located and governed by explicit, testable state transitions rather than ad hoc wiring.

11.2 Editor/AI Bridges

While the Enhanced IDE surface (Section 11.1) provides a developer-friendly multi-pane workspace, the clinically relevant behaviour emerges only when the AI assistant can *reliably* read from and write to the same editor buffers and project tree. This coupling is realised by a set of editor/AI bridges that mediate:

1. how editor state (file tree, active buffer, current selection) is transformed into an AI context window, and
2. how AI responses (free-text suggestions, structured action plans, or diff-style corrections) are translated back into file and buffer mutations under explicit safety constraints.

At a high level, let $\sigma_{\mathcal{E}}$ denote the editor state, $\sigma_{\mathcal{F}}$ the project file tree, μ the AI chat transcript, and p the current user prompt. The editor/AI bridging pipeline can be summarised as

$$(\sigma_{\mathcal{E}}, \sigma_{\mathcal{F}}, \mu, p) \xrightarrow{\Phi_{\text{ctx}}} c \xrightarrow{f_{\text{prov}}} r \xrightarrow{\Phi_{\text{plan}}} (\sigma'_{\mathcal{E}}, \sigma'_{\mathcal{F}}, \pi), \tag{11.2}$$

where c is the constructed context bundle, f_{prov} the provider-specific completion function (described in Section 4), r the raw AI response, and π an optional `ActionPlan` that encodes file edits. The maps Φ_{ctx} and Φ_{plan} are implemented across:

- `src/lib/ai/context.ts` (context building and token budgeting),
- `src/services/editorBridge.ts`, `src/services/editor/bridge.ts`, `src/services/editor/aiEditorBridgeGlobal.ts` (editor bridges and global glue), and
- `src/services/actions/schema.ts`, `src/services/actions/textdiff.ts`, `src/services/actions/runner.ts`, `src/lib/ai/diff.ts` (action plans, unified diffs, and patch application).

Figure 32 shows the vertical flow of these transformations.

11.2.1 Design goals and invariants

The editor/AI bridges are designed around three invariants that are central for clinical tooling:

Deterministic mapping. For a fixed editor state, file tree, AI configuration, and prompt, the context builder Φ_{ctx} is a pure function (up to timestamps in fenced comments). This enables reproducible AI interactions, which is crucial when a psychiatry trainee and supervisor are reviewing the same refactoring episode.

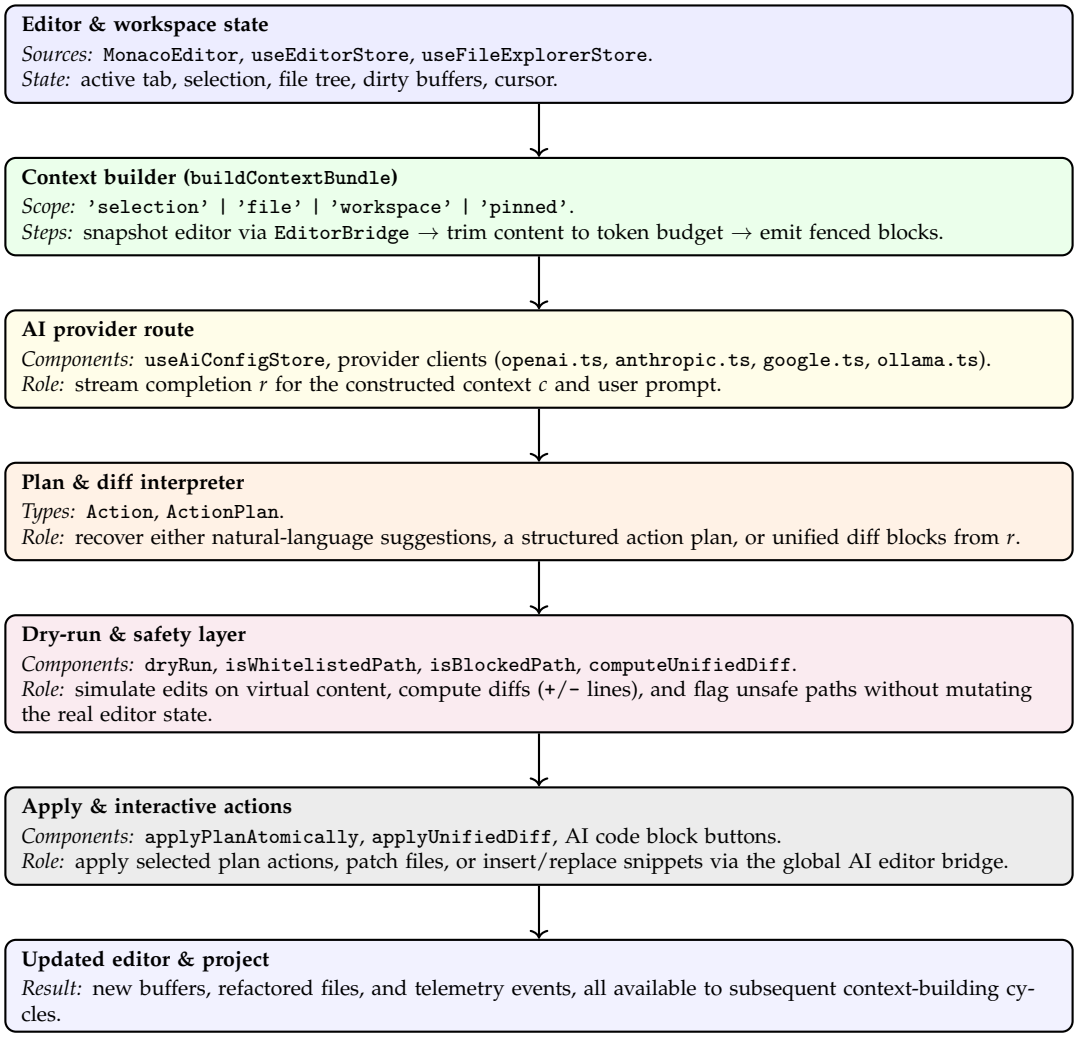


Figure 32: *Vertical schema of the editor/AI bridge. Editor state flows downward to the AI provider as a token-budgeted context bundle and returns as either a structured action plan or diff blocks. A dry-run layer and global editor bridge ensure that all mutations are explicit, inspectable, and reversible.*

Safety by construction. The plan executor Φ_{plan} never edits arbitrary paths. Instead, the `isWhitelistedPath` and `isBlockedPath` predicates in `src/services/actions/safety.ts` enforce a coarse-grained path policy:

$$\texttt{WHITELIST} = \{\, p : p \text{ starts with } \texttt{"src/"}, \texttt{"apps/"}, \texttt{"packages/"}, \texttt{"tests/"} \,\},$$

while `BLOCKED` targets secrets (e.g. `.env`, private keys). Any proposed action outside the whitelist or matching a blocked pattern is flagged and excluded from application.

Undoability. All editor mutations are routed through the same `useEditorStore` history mechanism used for manual edits (via helper functions such as `pushHistory`). AI-triggered changes can therefore be undone using the ordinary undo stack, mirroring psychiatric practice where every intervention should be reversible or, at minimum, auditable.

11.2.2 Context construction and selection scopes

The transformation Φ_{ctx} is implemented in `src/lib/ai/context.ts`. The module exposes:

- the scope enumeration

$$\texttt{Scope} = \texttt{'selection'} \mid \texttt{'file'} \mid \texttt{'workspace'} \mid \texttt{'pinned'},$$

- an `EditorBridge` interface used for live snapshots, and
- the context builder `buildContextBundle(opts): BuiltContext`.

The `EditorBridge` is defined as

```
export interface EditorBridge {
  getActiveFilePath?: () => string | undefined;
  getActiveFileContent?: () => string | undefined;
  getSelection?: () => {
    text: string;
    path?: string;
    startLine?: number;
    endLine?: number;
  } | undefined;
}
```

and is implemented at runtime in `src/services/editor/aiEditorBridgeGlobal.ts` by reading from `useEditorStore` and `useFileExplorerStore`. This global bridge exposes a minimal set of read operations to the AI context layer without embedding React components into the AI pipeline.

The core budgeting logic in `buildContextBundle` can be expressed as:

$$r_{\text{res}} = \min\big(512,\ \lfloor 0.1 \cdot B \rfloor\big), \tag{11.3}$$

$$b_{\text{ctx}} = \max\big(512,\ B - R - r_{\text{res}}\big), \tag{11.4}$$

where B is the requested token budget, R the reserved completion window (`responseTokens`), and b_{ctx} the effective context budget. If $b_{\text{ctx}} \leq 0$, the function returns a short policy stub explaining that context has been omitted. Otherwise, it greedily packs content segments into fenced blocks until the budget is exhausted.

Table 51 summarises how each scope translates editor state into context content.

Inside the builder, content slices are wrapped using a simple fencing convention:

```
<<<FILE path="src/features/psychiatry/flows/AgitationFlowShell.tsx">>
...code...
<<<END FILE>>>
```

This convention is recognised both by prompt templates and by downstream parsers, ensuring that AI models can reliably refer to specific files and that structured responses (e.g. action plans) can reference paths without ambiguity.

11.2.3 Action plans and diff-style suggestions

On the response side, the bridge distinguishes between unstructured suggestions and structured *action plans*. The latter are encoded by the `Action` and `ActionPlan` types in `src/services/actions/schema.ts`:

Table 51: *Context scopes used by `buildContextBundle` and their relationship to editor state.*

Scope	Editor source	Typical usage
`'selection'`	Active selection in the current Monaco tab, together with optional line numbers and path from `getSelection()`.	Focused refactoring or explanation of a small code or configuration fragment, analogous to zooming into a single symptom cluster in a mental state examination.
`'file'`	Entire content of the active file from `getActiveFileContent()`, capped with *head/tail* trimming when the file exceeds the budget.	Whole-file refactoring, documentation generation, or safety review of a specific module (e.g. one clinical flow or one psychometric scoring function).
`'workspace'`	A curated pack of files from the project tree, including open tabs and nearby configuration files, each fenced as a separate block.	Cross-file tasks such as aligning flows and prompts, updating multiple psychometric scales, or coordinating IDE theme tokens with clinical card palettes.
`'pinned'`	Manually pinned attachments or reference snippets passed in via `pinned` or `attachments` options.	Including clinical guidelines, hospital protocols, or paper excerpts as non-editable context when requesting AI transformations on the codebase.

```
export type Path = string;

export type Action =
  | { kind: 'create';  path: Path; text: string; language?: string }
  | { kind: 'replace'; path: Path; text: string }
  | { kind: 'modify';  path: Path;
      edits: { range: [number, number, number, number]; text: string }[] }
  | { kind: 'rename';  from: Path; to: Path }
  | { kind: 'delete';  path: Path }
  | { kind: 'format';  path: Path; language?: string };

export type ActionPlan = {
  id: string;
  title: string;
  createdAt: number;
  actions: Action[];
  note?: string;
};
```

From a formal standpoint, each action $a \in \texttt{Action}$ induces a partial function on file contents:

$$\varphi_a : \mathcal{S} \rightharpoonup \mathcal{S},$$

where $\mathcal{S}$ is the set of all finite strings. For example, a `replace` action with payload (p, t) maps the old content s_p of path p to the new content t, while a `modify` action with edits $\{e_i\}$ applies a finite set of non-overlapping range replacements to s_p in reverse order of their starting offsets.

Table 52 provides an operational summary of the supported action kinds.

Table 52: Summary of `Action` variants and their operational effect on file contents.

Action kind	Key fields	Operational effect
`'create'`	`path`, `text`, optional `language`	Creates a new file at `path` with content `text`, using the language hint to configure editor syntax highlighting.
`'replace'`	`path`, `text`	Replaces the entire content of the file at `path` with `text`. Used for relatively small files or for regenerating configuration templates.
`'modify'`	`path`, list of `edits`	Applies multiple range-based edits in a single pass, preserving unaffected lines. Particularly suitable for targeted refactoring (e.g. inserting a new hook without rewriting the whole file).
`'rename'`	`from`, `to`	Renames a file while preserving its content and editor history, updating references in the file tree.
`'delete'`	`path`	Removes the file at `path` from the project tree. In the current implementation, this is only allowed on whitelisted paths and is surfaced prominently in the UI.
`'format'`	`path`, optional `language`	Rewrites the file contents using a formatter consistent with the detected or specified language (e.g. TypeScript or JSON), typically preserving semantics.

For non-plan responses, the assistant often emits unified diff blocks or plain code fences. Unified diffs are constructed and interpreted via `src/services/actions/textdiff.ts` and `src/lib/ai/diff.ts`. Given a path p and before/after contents s_{old}, s_{new}, the helper `computeUnifiedDiff` emits a diff string of the form

$$d = \texttt{"--- "}\, p \parallel \texttt{"+++ "}\, p \parallel \ell_1 \parallel \ldots \parallel \ell_n,$$

where each ℓ_i is either an unchanged line, a deletion (leading `-`), or an insertion (leading `+`). The helper `countDiffLines` then computes

$$\text{added}(d) = \#\{\ell_i : \ell_i \text{ starts with } \texttt{+}\}, \tag{11.5}$$

$$\text{removed}(d) = \#\{\ell_i : \ell_i \text{ starts with } \texttt{-}\}, \tag{11.6}$$

which is surfaced in the UI as a quick measure of patch magnitude.

11.2.4 Dry-run semantics and safety filters

The `dryRun` function in `src/services/actions/runner.ts` implements a non-destructive simulation of a candidate `ActionPlan`. On input π, it iterates through the plan's actions, checks path-level policies using `isWhitelistedPath` and `isBlockedPath`, and then computes virtual before/after contents plus diffs without writing to disk.

The resulting structure is:

```
export type DryRunResult = {
  file: string;
```

```
  before: string;
  after: string;
  diff: string;
  added: number;
  removed: number;
  blocked?: boolean;
};
```

and the dry-run function has type

$$\texttt{dryRun} : \texttt{ActionPlan} \longrightarrow \texttt{DryRunResult}[\,],$$

i.e. it returns an array of per-file outcomes.

Operationally, an action on path p is processed as follows:

1. If p fails `isWhitelistedPath(p)` or matches `isBlockedPath(p)`, the action is marked `blocked` and skipped for the purposes of computing `before`, `after`, and `diff`.
2. Otherwise, the current content s_{old} for p is obtained via the editor bridge (typically using the file explorer store as backing). The action is simulated to yield $s_{\text{new}} = \varphi_a(s_{\text{old}})$.
3. A unified diff $d = \texttt{computeUnifiedDiff}(p, s_{\text{old}}, s_{\text{new}})$ is computed and summarised by `countDiffLines`.

The IDE surfaces this information in the AI panel as a plan inspector, where each planned change can be expanded, inspected, and selectively accepted or rejected. In clinical terms, this mirrors the distinction between formulating an intervention plan and actually changing a patient's prescription: the dry run allows a psychiatrist–developer to visualise potential consequences before commiting to the refactor.

When the user confirms a subset of actions, the executor (e.g. `applyPlanAtomically`) applies them in a transaction-like manner, coordinating with `useEditorStore` so that all edits propagate into the Monaco editor and its undo stack.

11.2.5 Global AI editor bridge and interactive code blocks

In addition to structured action plans, the AI assistant frequently emits plain code blocks or diff snippets inside the chat transcript. To make these immediately actionable, the system exposes a global AI editor bridge in `src/services/editor/aiEditorBridgeGlobal.ts`:

```
declare global {
  interface Window {
    __AI_EDITOR_BRIDGE__?: {
      insertAtCursor?: (c: string) => void;
      replaceSelection?: (c: string) => void;
      writeFile?: (p: string, c: string,
        opts?: { create?: boolean; overwrite?: boolean }) => void;
      readFile?: (p: string) => string | undefined;
      fileExists?: (p: string) => boolean;
      showToast?: (msg: string,
        level?: 'info'|'error'|'success'|'warn') => void;
    };
  }
}
```

On the producer side, this bridge is wired to the React stores via `buildBridge()`, which uses `useFileExplorerStore` and `useEditorStore` to implement the above methods, together with `inferLanguageFromFence` for untitled file creation. On the consumer side, components such as `src/components/ai/CodeBlockWithActions.tsx` and `src/components/ai/panel/CodeBlock.tsx` expose concrete affordances:

- **Insert.** Calls `insertAtCursor` with the block content, falling back to clipboard copy if no editor is available.
- **Replace.** Calls `replaceSelection` when a selection exists, otherwise falls back to *Insert*.
- **New.** Infers a plausible file name and extension from the fence info (via `mapFenceToLangAndExt`) and delegates to `writeFile` to create a new file under the project root (typically `src/`).
- **Apply diff.** For unified diff blocks, uses `applyUnifiedDiff` from `src/lib/ai/diff.ts`, passing the bridge's `readFile` and `writeFile` functions.

These operations are all local and synchronous with respect to the editor stores; they do not themselves contact any external AI provider. In practice, this means that once an AI response has arrived, the user can experiment with multiple insertion and replacement strategies, undo them, and iterate, all without incurring additional network calls.

In combination, the context builder, action-plan executor, and global AI editor bridge turn the IDE into a bidirectional interface: the AI assistant gains a structured, budget-aware view of the workspace, and the psychiatrist–developer receives refactorings and suggestions that are grounded in that view, surfaced as explicit, reversible edits rather than opaque, one-shot patches.

11.3 Telemetry and Logging

In the Enhanced IDE, telemetry and logging serve as a bridge between the AI-assisted editing surface, the embedded terminal, and the observability infrastructure described in Section 12. Rather than treating logs as an implementation detail confined to the browser console, SynapseCore promotes a structured, typed log stream that can be consumed both by human users (through the IDE terminal) and by downstream analytics components. This subsection describes the terminal log bus abstraction, its integration with task execution and apply-plan workflows, and the way in which AI actions and token usage can be summarised for clinical-informatics users.

11.3.1 Terminal log bus abstraction

The IDE exposes a lightweight logging infrastructure centred on the `terminalLogBus` module (`src/components/terminal/terminalLogBus.ts`). At the type level, each log emitted to the bus is represented as a small, structured record:

```
export interface TerminalSyntheticLog {
  channel: 'latex' | 'system' | 'build';
  level: 'info' | 'success' | 'error';
  message: string;
  timestamp: Date;
}
```

The `channel` field partitions logs into conceptual streams: `'latex'` for events related to LaTeX compilation and document generation, `'system'` for general IDE messages, and

Table 53: *Terminal log channels and their intended semantic scope in the Enhanced IDE.*

Channel	Typical source	Clinical / developer interpretation
`latex`	LaTeX export and compilation helpers (e.g. PDF preview tasks)	Indicates success or failure of document-generation steps linked to clinical artefacts such as letters, reports, or teaching material.
`system`	IDE shell, file explorer, editor bridge, internal scripts	General-purpose messages about navigation, file operations, or AI-initiated actions; useful for debugging apply-plan steps or editor automation.
`build`	Task bridge and simulated build system (`tasksBridge.ts`)	Represents run/build tasks triggered from the IDE, providing feedback about compilation, bundling, or external VS Code tasks.

`'build'` for simulated or external build tasks. The `level` reflects coarse-grained severity (`'info'`, `'success'`, `'error'`), while `message` is a human-readable description and `timestamp` captures the time of emission.

Abstractly, let

$$\mathcal{C} = \{\text{latex}, \text{system}, \text{build}\}, \quad \Lambda = \{\text{info}, \text{success}, \text{error}\},$$

and let $\mathcal{M}$ denote the set of possible messages (Unicode strings) and $\mathbb{T}$ the time domain. A single log entry is then a quadruple

$$\ell = (c, \lambda, m, \tau) \in \mathcal{C} \times \Lambda \times \mathcal{M} \times \mathbb{T}.$$

The terminal bus maintains a set of subscribers $\mathcal{S} = \{s_1, \ldots, s_n\}$, where each subscriber is a callback $s_i : \mathcal{C} \times \Lambda \times \mathcal{M} \times \mathbb{T} \to \varnothing$. When a new log ℓ is emitted, the bus applies ℓ to all registered listeners:

$$\text{emit}(\ell) = (s_1(\ell), s_2(\ell), \ldots, s_n(\ell)).$$

The implementation mirrors this mathematical view. Subscribers are registered using `subscribeTerminalLogs(listener: TerminalLogListener)`; internally, the module keeps a `Set<TerminalLogListener>` and exposes a private `emitTerminalLog` helper that applies each listener in turn. For ease of use, three convenience functions are exported:

```
export const terminalInfo = (message: string,
  channel: TerminalSyntheticLog['channel'] = 'system') =>
  emitTerminalLog({ channel, level: 'info', message });

export const terminalSuccess = (message: string,
  channel: TerminalSyntheticLog['channel'] = 'system') =>
  emitTerminalLog({ channel, level: 'success', message });

export const terminalError = (message: string,
  channel: TerminalSyntheticLog['channel'] = 'system') =>
  emitTerminalLog({ channel, level: 'error', message });
```

These helpers ensure that higher-level components (e.g. the tasks bridge or LaTeX export workflows) can log semantic events without depending on the internal representation of listeners. From the perspective of the overall system, the terminal log bus is therefore a small, typed event hub that specialises in human-readable diagnostic lines rather than raw traces.

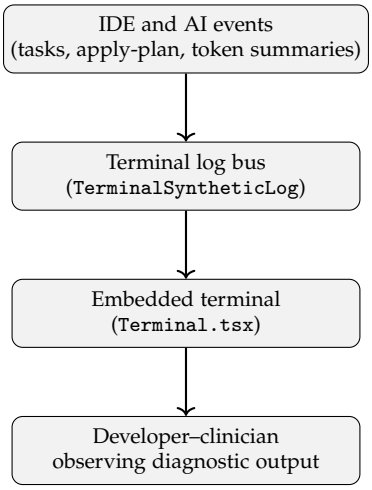


Figure 33: *Vertical schema of the terminal log pipeline in the Enhanced IDE. Structured events from IDE and AI subsystems are normalised into* `TerminalSyntheticLog` *entries and rendered in the embedded terminal for human inspection.*

The terminal component (`src/components/terminal/components/Terminal.tsx`) subscribes to the log bus during mount. Each `TerminalSyntheticLog` is converted into a `TerminalCommand` with a rendered "fake" command name (e.g. `[build]` or `[system]`) and an output line of the form `LEVEL: message`. This ensures that logs appear in the same scrollable history as user-entered commands, preserving chronological order and context.

11.3.2 Bridging tasks, terminal output, and apply-plan activity

A central design goal of the Enhanced IDE is to keep AI-assisted operations and build-like tasks transparent. To this end, the `tasksBridge` and `tasksAdapter` modules (`src/services/tasksBridge.ts` and `src/services/tasksAdapter.ts`) connect higher-level IDE actions to the terminal log bus.

The `tasksBridge` maintains an optional handler for task execution:

```
type TaskKind = 'run' | 'build';

let handler: null | ((kind: TaskKind) => void) = null;

export function setTasksHandler(fn: ((kind: TaskKind) => void) | null) {
  handler = fn;
}

export function triggerTask(kind: TaskKind) {
  if (handler) {
    handler(kind);
  } else {
    terminalInfo(
      'Build: Compiling project (simulated)...',
      'build'
    );
  }
}
```

If a real task handler is registered (for instance by an embedded VS Code environment via `tasksAdapter`), `triggerTask` will delegate to that handler. Otherwise, it emits a synthetic `build` log through `terminalInfo`, ensuring that users still receive feedback when "Run" or "Build" are invoked.

The `tasksAdapter` module is designed to run in contexts where a VS Code-like host is present. It introspects the global `window.acquireVsCodeApi` and, if available, registers a custom handler that forwards `'run'` and `'build'` requests to the host. On success or error, the adapter emits logs via `terminalInfo` or `terminalError`, respectively. In effect, this makes task-related telemetry visible inside the web-based SynapseCore terminal even when the underlying compilation or test execution is delegated to an external process.

From an abstract perspective, let $\mathcal{K} = \{\text{run}, \text{build}\}$ denote the set of task kinds and let $\text{task} : \mathcal{K} \to \{\text{handled}, \text{fallback}\}$ encode whether a given invocation is served by an external handler. Whenever a task $k \in \mathcal{K}$ is triggered, the IDE emits a log

$$\ell_k = \begin{cases} (\text{build}, \text{info}, m_k, \tau_k), & \text{if task}(k) = \text{fallback}, \\ (\text{build}, \lambda_k, m_k, \tau_k), & \text{if task}(k) = \text{handled}, \end{cases}$$

where m_k is a short, human-readable description and λ_k is a level derived from the external outcome (success or error). In both cases, the resulting synthetic log is consumed by the terminal subscriber and rendered as a new line in the history.

Although the apply-plan execution logic (`executeApplyPlan` in `src/utils/ai/apply/executeApplyPlan.ts`) currently logs to the development console, it is intentionally designed to be compatible with this terminal bus. Each apply-plan run has a well-defined set of actions (create, replace, insert) and a finite collection of target paths. A natural extension—and one that fits the existing design—is to emit a `system` log summarising the plan:

$$m_{\text{plan}} = \text{"Apply-plan executed: } n_{\text{files}} \text{ files touched, } n_{\text{create}} \text{ created, } n_{\text{replace}} \text{ replaced."}$$

where the counts are computed from the `ApplyPlan` object. This kind of message can be routed through `terminalSuccess` on success, or `terminalError` if any file operation fails. Such instrumentation makes AI-driven modifications auditable from within the IDE itself, an important requirement when the same environment is used for clinical and educational content.

11.3.3 AI actions, token usage, and local observability

While the terminal log bus primarily targets human-readable messages, SynapseCore simultaneously maintains a more fine-grained telemetry stream for AI activity. The generic AI telemetry emitter (`src/components/ai/telemetry/events.ts`) accepts arbitrary `TelemetryEvent` records and queues them in a global `window.telemetry.queue`, optionally echoing them to the console in development builds. Downstream, the observability layer (`src/observability/otel.ts`) uses the `METRICS` object to record histograms and counters such as request latency (`reqLatencyMs`), prompt and completion tokens (`promptTok`, `complTok`), and inferred cost (`costUsd`).

If we denote by T_{prompt} and $T_{\text{completion}}$ the number of prompt and completion tokens reported by the AI provider for a single request, and by c_{prompt} and $c_{\text{completion}}$ the per-token prices, then the estimated cost for that request can be modelled as

$$\text{cost} = T_{\text{prompt}} \cdot c_{\text{prompt}} + T_{\text{completion}} \cdot c_{\text{completion}}.$$

These quantities are accumulated over time by the counters in METRICS, providing a running estimate of usage and expense across sessions and providers.

From an IDE perspective, the crucial point is that the same information can be summarised and surfaced in the terminal as synthetic logs. A typical message might be of the form

$m_{\text{ai}} =$ "AI route (p, m) used T_{prompt} prompt tokens and $T_{\text{completion}}$ completion tokens (cost USD)."

Emitting such summaries via terminalInfo on the system channel provides developers and clinician-informaticians with a fine-grained sense of how expensive or intensive their AI workflows are without requiring them to open a separate metrics dashboard.

Finally, the local logger (src/lib/logger.ts) prefixes internal console messages with [AI-TRACE] and a monotonic timestamp, offering a low-level trace that complements both the structured telemetry events and the terminal log stream. Together, these layers create a spectrum of observability:

- *Console-level traces* (logger) capture detailed debugging information for developers working on the AI pipeline.
- *Metrics-level counters and histograms* (METRICS) support aggregate analysis of latency, token usage, and cost.
- *Terminal-level synthetic logs* (terminalLogBus, Terminal.tsx) expose key AI and task-related events in a human-friendly, clinically situated interface.

For clinical psychiatry professionals who also act as maintainers of the workbench, this layered approach makes it possible to reason about AI behaviour, resource usage, and automation effects directly within the IDE context, without sacrificing formal metrics or downstream observability.

11.4 Use Cases for Clinical Informatics

The Enhanced IDE and psychiatry modules are not only developer-facing artifacts but also a substrate for clinical informatics work. In practice, informatics leads, academically oriented clinicians, and trainees can use the same workbench to prototype new flows, extend measurement-based care coverage, and assemble institution-specific content packs that remain versioned and auditable. This subsection illustrates these use cases and connects them to concrete types and files in the codebase.

Table 54 summarises three recurring patterns: rapid creation of flows for specific protocols, custom scale calculators and prompts for measurement-based care, and institution- specific content packs managed through a light-weight versioning layer.

Figure 34 shows a vertical schema of how clinical informatics work flows from protocol design into concrete SynapseCore artefacts and, ultimately, into bedside documentation.

11.4.1 Rapid creation of new flows

The flows subsystem in centerpanel/Flows encodes complex clinical processes (e.g. agitation de-escalation, capacity assessment, catatonia workup) as sequences of strongly typed steps with outcome objects. For the clinical informatics user, the key point is that new flows can be prototyped rapidly by reusing the existing shells and builder conventions.

At the type level, the file flowTypes.ts introduces the FlowId union (e.g. "safety", "agitation", "capacity", "observation"), and the StepPill structure for labelling navigation pills. Each concrete flow shell in centerpanel/Flows/shells (such as SafetyFlowShell.tsx

Table 54: *Representative clinical informatics use cases in SynapseCore, with the main technical touchpoints and clinical outputs.*

Use case	Primary modules	Clinical output
Rapid creation of new flows	`centerpanel/Flows/shells`, `centerpanel/Flows/builders`, `centerpanel/Flows/flowTypes.ts`	Structured, multi-step pathways (e.g. agitation, capacity, observation) encoded as reusable flow shells with outcome objects and AI-ready summaries.
Custom scale calculators and prompts	`features/psychiatry/mbc/calculators.ts`, `features/psychiatry/seeds/psychometrics.ts`, `components/ai/psychiatry/prompts.ts`	Deterministic autoscore functions and tailored AI prompts for depression, anxiety, PTSD, OCD, substance use, and locally defined instruments.
Institution-specific content packs with version control	`features/psychiatry/content/section8.index.ts`, `features/psychiatry/content/assembleSlice.ts`, `features/psychiatry/seeds/*`, IDE integration	Packaged instruments and guidance (e.g. AUDIT-C, PSQI, local risk policies) with validators, cached slices, and modifiable cards that can be iterated via the IDE and AI apply-plan system.

or `AgitationFlowShell.tsx`) implements a fixed sequence of steps, renders the appropriate form components, and ultimately produces an outcome object whose shape is defined in the corresponding builder file (e.g. `agitationOutcome.ts`, `capacityOutcome.ts`).

Formally, a flow can be viewed as a finite state machine

$$\mathcal{F} = (S, s_0, S_{\text{term}}, \delta, \Omega),$$

where S is the set of step identifiers (e.g. `"triage"`, `"de-escalation"`, `"outcome"`), $s_0 \in S$ is the initial step, $S_{\text{term}} \subseteq S$ the set of terminal steps, $\delta : S \times A \to S$ a transition function for actions (e.g. "Next", "Back", "Skip"), and Ω a mapping from terminal states to outcome objects. In the implementation, $\Omega(s_{\text{term}})$ is constructed by the builder helpers in `centerpanel/Flows/builders`, assembling a structured JSON record summarising risk, interventions, and follow-up plans.

From a clinical informatics perspective, rapid creation of a new flow (e.g. an early psychosis pathway or an ECT pre-assessment flow) proceeds roughly as:

1. *Define the step skeleton*: specify the set S and an ordering for the step pills in a new shell component (e.g. `EarlyPsychosisFlowShell.tsx`), reusing the layout primitives from existing shells.
2. *Specify the outcome schema*: design a TypeScript type for the outcome object and implement a builder (e.g. `earlyPsychosisOutcome.ts`) that maps form state into that type. This guarantees that AI and export modules see a stable schema.
3. *Register the flow*: add a new `FlowId` branch in `flowTypes.ts` and update `FlowHost.tsx` and `flowLibraryMeta.ts` so that the flow appears in the library view and can be launched from the shell.
4. *Integrate with AI*: extend the context-building logic so that the new outcome object is serialised into the AI prompt for risk summaries, letters, or supervision notes.

At runtime, the `FlowHost` component switches between active flows based on the selected `FlowId`, using `useFlowsUIStore` for state. The same host also supports a *run review* mode, in which previously completed flows are rendered read-only (e.g. via `CompletedRunReviewShell.tsx`). This makes it straightforward to compare flow implementations across institutions or versions, as old runs continue to use older schemas while new runs adopt updated ones.

Because shells and builders are written in idiomatic React/TypeScript, they can be gen-

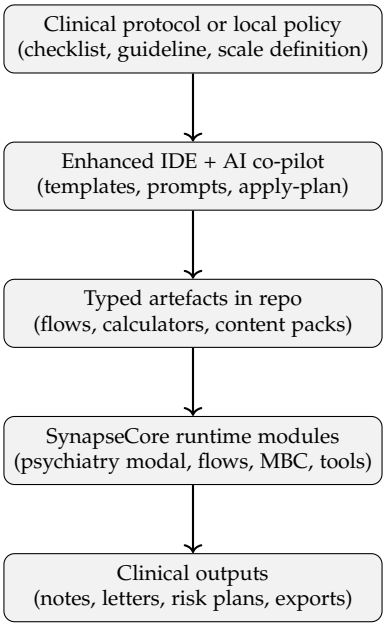


Figure 34: *Vertical overview of clinical informatics activities in SynapseCore. Human-authored protocols are translated into typed artefacts using the Enhanced IDE and AI co-pilot, then surfaced to clinicians via the psychiatry, flows, MBC, and tools modules.*

erated or refactored using the Enhanced IDE with AI co-pilot support. In practice, a clinical informatics engineer can prototype a flow in natural language, ask the AI to draft a skeleton shell and outcome builder, then validate and refine the code before exposing the new flow to clinicians.

11.4.2 Custom scale calculators and prompts

Measurement-based care is implemented by the MBC engine in `features/psychiatry/mbc/calculators.ts`, which defines a generic `ScoreResult` and a set of scale-specific scoring functions such as `phq9Score`, `gad7Score`, `pcl5Score`, and others. For each scale, the engine ensures that input vectors are clamped and coerced to the appropriate length and range, and that severity bands and flags are computed deterministically from the total score.

Abstractly, a scale k is represented as a tuple

$$\mathcal{S}_k = (n_k, \mathcal{X}_k, f_k, \mathcal{B}_k, \mathcal{F}_k),$$

where n_k is the number of items, $\mathcal{X}_k \subseteq \mathbb{Z}^{n_k}$ the allowed response space (e.g. $\{0,1,2,3\}^9$ for PHQ-9), $f_k : \mathcal{X}_k \to \mathbb{N}$ a total-score function, $\mathcal{B}_k$ a finite set of severity bands (with lower and upper bounds), and $\mathcal{F}_k$ a set of flags (e.g. item-level risk signals). The concrete implementation is given by a function `scaleScore(items: number[]): ScoreResult` that returns `total`, `severity`, `bands`, and `flags`.

The psychometrics seed file `features/psychiatry/seeds/psychometrics.ts` declares cards such as `"PHQ-9 -- Depression Severity (Autoscore)"`, `"GAD-7 -- Anxiety Severity (Autoscore)"`, `"PCL-5 -- PTSD Symptom Severity (Autoscore)"`, and many others, each tagged with identifiers like `"phq9"`, `"gad7"`, `"pcl5"`, and associated with leaf IDs indicating recommended frequency (e.g. biweekly, weekly, monthly). These cards serve as entry points in the psychiatry modal, linking specific instruments to autoscore calculators and to AI prompts.

From a clinical informatics standpoint, adding a new instrument (say, a localised trauma scale) involves the following steps:

1. *Implement the scoring function* in `calculators.ts`, following the existing pattern: coerce inputs to the appropriate length and range, compute the total score, derive severity labels using a band array, and set any clinically relevant flags.
2. *Create or extend psychometrics cards* in `psychometrics.ts`, referencing the new scale via tags and ensuring that section IDs and frequency (e.g. weekly, monthly) match institutional policy.
3. *Update AI prompts* in `components/ai/psychiatry/prompts.ts` so that risk and treatment narratives explicitly incorporate the new scale, including item anchors and interpretive guidance.
4. *Optionally expose instructions* (e.g. patient-facing text) via inline or referenced text sources in the content system, ensuring that clinicians can view or copy the instructions from within the workbench.

If the scale is sensitive (e.g. a suicidality screen), flags and bands can be chosen so that the autoscore engine emits explicit `flags` that drive AI prompts into a more guarded mode (for example, prioritising crisis resources and recommending immediate in-person review). In future work, these flags can also be fed into the risk-grade mapping described in the MBC formalisation section of this manuscript.

11.4.3 Institution-specific content packs with version control

SynapseCore explicitly supports institution-specific content packs, implemented in the `features/psychiatry/content` layer. The `section8.index.ts` and `section8.registry.ts` files illustrate this pattern for a set of common instruments such as PSQI, GHQ-12, EDE-Q, AUDIT, AUDIT-C, DAST-10, ASSIST, ASRS, Vanderbilt scales, SNAP-IV, and the C-SSRS. Each content pack includes rich metadata: overview text, lists of clinical uses, exemplar forms, prompts, and bibliographic references.

A minimalised view of such a pack is

$$P = (\text{id}, \text{overview}, \text{clinicalUse}, \text{exampleForm}, \text{prompts}, \text{references}, \text{licensingNote}, v),$$

where $v \in \mathbb{N}$ is a version-like field used in cache keys, and the remaining components correspond to the fields validated by `assertContentPack`. The helper `assertContentPack` checks for completeness (e.g. at least three clinical uses, a minimum number of prompts and references, and a licensing note if required) and logs warnings for incomplete content.

When cards are rendered, the `assembleSlice.ts` module builds an `EvidenceSlice` by resolving text, prompts, and references via `resolveTextSource`, `resolvePromptsSource`, and `resolveReferencesSource`. To avoid unnecessary recomputation, each slice is cached by a key of the form

$$k_{\text{card}} = \texttt{card.id} \,\|\, (\texttt{card.updatedAt} \text{ or } \texttt{card.version}),$$

stored in a `Map<string,EvidenceSlice>`. Whenever a card is modified in the underlying TypeScript or content files (for example, when an institution updates its local AUDIT-C guidance), the `updatedAt` or `version` field should be bumped, forcing a cache miss and recomputation of the slice. This provides a simple but effective form of version control at the content level.

In practice, clinical informatics teams can maintain one or more institution-specific packs (e.g. a "Section 8" folder per hospital network) with:

- *Local risk policies*, such as thresholds for triggering mandatory review, and recommended wording for safety plans.
- *Institutional scales or variants*, for example site-specific adaptations of GHQ-12 or sleep diaries.
- *Locale-specific content*, including Turkish and English variants, by leveraging the `LocaleCode` abstraction.

These packs can be edited using the Enhanced IDE, with the AI co-pilot assisting in drafting overview text, clinical use bullet points, or example forms. Because the content schema is typed, the apply-plan system can safely insert or replace content blocks while preserving the overall structure required by `assertContentPack`. Combined with telemetry and the terminal logging described in Section 11.4, this enables an iterative, auditable workflow in which institutional content evolves alongside clinical practice, yet remains embedded in a single, transparent codebase.

12 Observability and Telemetry

12.1 OpenTelemetry Integration

The observability layer of SynapseCore is designed to be compatible with OpenTelemetry (OTEL) while remaining lightweight enough to run entirely in the browser. Rather than hard-coding a particular tracing backend, the integration in `observability/otel.ts` and `observability/spans.ts` assumes that an external host (for example, a VS Code extension or an institution-specific wrapper) will provide an OTEL tracer and meter via the global `window` object. When this instrumentation is present, the application emits traces and metrics for AI calls, orchestration rounds, and other high-value events; when it is absent, the system falls back to no-op implementations and continues to function without telemetry.

12.1.1 Bootstrap and idempotent initialisation

The file `otel.ts` exposes a single entry point, `initOtelOnce()`, which is called during application start-up. This function is responsible for wiring up the tracer and meter only when observability is enabled and only once per browser session.

At module load time, two conservative defaults are defined:

```
let tracer: any = {
  startSpan: (_name: string) => ({ setAttribute() {}, end() {} })
};
let meter: any = {
  createCounter: () => ({ add() {} }),
  createHistogram: () => ({ record() {} })
};
```

These no-op objects guarantee that calls to `getTracer()` and `getMeter()` remain safe even if no external OTEL implementation is injected.

The `initOtelOnce()` function then performs three checks:

1. *Feature flags*: if neither `CONFIG.flags.enableTracing` nor `CONFIG.flags.enableMetrics` is true, the function returns immediately and no further set-up occurs.

2. *Idempotence*: a guard flag `window.__otel_inited` is used to ensure that initialisation runs at most once. If the flag is already set, the function returns without modifying anything.
3. *Environment probes*: the function attempts to read `window.otelTracer` and `window.otelMeter`. If they exist, they replace the no-op defaults. In addition, if a function `window.__otel_setup` is present, it is invoked to allow a host environment to perform any additional configuration (for example, wiring exporters, service names, or resource attributes).

Formally, let T_0 and M_0 be the default no-op tracer and meter, and let T_{env} and M_{env} denote any tracer and meter supplied by the host environment via `window`. The effective tracer T and meter M after initialisation are

$$\mathsf{T} = \begin{cases} \mathsf{T}_{\text{env}}, & \text{if telemetry enabled and } \mathsf{T}_{\text{env}} \text{ is available,} \\ \mathsf{T}_0, & \text{otherwise.} \end{cases}$$

$$\mathsf{M} = \begin{cases} \mathsf{M}_{\text{env}}, & \text{if telemetry enabled and } \mathsf{M}_{\text{env}} \text{ is available,} \\ \mathsf{M}_0, & \text{otherwise.} \end{cases}$$

The accessor functions `getTracer()` and `getMeter()` simply return T and M, thereby hiding the presence or absence of a true OTEL provider from the rest of the code.

12.1.2 Metrics map and semantic counters

Beyond tracing individual spans, the observability layer exposes a small, fixed set of metrics for AI and tooling. These metrics are collected in the exported `METRICS` object in `otel.ts`:

```
export const METRICS = {
  reqLatencyMs: meter.createHistogram?.('req_latency_ms') ?? { record() {} },
  promptTok:   meter.createCounter?.('tokens_prompt')     ?? { add() {} },
  complTok:    meter.createCounter?.('tokens_completion') ?? { add() {} },
  costUsd:     meter.createCounter?.('cost_usd')          ?? { add() {} },
  errors:      meter.createCounter?.('errors_total')      ?? { add() {} },
  rlHits:      meter.createCounter?.('rate_limit_hits')   ?? { add() {} },
  cacheHits:   meter.createCounter?.('cache_hits')        ?? { add() {} },
};
```

Each field either points to a live OTEL metric instrument (if the meter was successfully initialised) or to a no-op stub with the appropriate method. This allows the rest of the application to call `record()` or `add()` unconditionally.

Table 55 summarises the semantics of these instruments.

To reason about these metrics more formally, consider an AI request indexed by i with prompt tokens P_i, completion tokens C_i, and estimated cost K_i (in USD). Over a time window $[0, T]$, the OTEL counters implement empirical aggregates

$$\text{Tok}^{\text{prompt}}(T) = \sum_{i : t_i \le T} P_i, \qquad \text{Tok}^{\text{compl}}(T) = \sum_{i : t_i \le T} C_i, \qquad \text{Cost}(T) = \sum_{i : t_i \le T} K_i,$$

while the histogram `req_latency_ms` approximates the empirical distribution of latency values $\{\ell_i\}$ for traced operations. These aggregates can be exported to typical OTEL backends (e.g. Prometheus-compatible systems, tracing dashboards) to support institutional monitoring of AI usage in clinical workflows.

Table 55: OpenTelemetry metric instruments exposed by the `METRICS` object and their intended meaning in the context of AI-assisted clinical tooling.

Instrument	Type	Clinical / operational interpretation
`req_latency_ms`	Histogram (ms)	End-to-end latency for traced operations (e.g. AI rounds, apply-plan executions). Useful for detecting slow model routes or environmental bottlenecks.
`tokens_prompt`	Counter (count)	Total number of prompt tokens sent to AI providers; allows estimation of utilisation and cost per unit time.
`tokens_completion`	Counter (count)	Total number of completion tokens received from providers; together with prompt tokens, approximates workload intensity.
`cost_usd`	Counter (currency)	Cumulative estimated monetary cost of AI usage, enabling monitoring of per-session and per-provider expenditure.
`errors_total`	Counter (count)	Number of traced operations that ended with an error; a rise may signal provider instability or malformed prompts.
`rate_limit_hits`	Counter (count)	Count of rate-limit responses from AI providers or other external services, important for capacity planning.
`cache_hits`	Counter (count)	Number of successful results served from local or remote caches, indicating effectiveness of caching strategies.

12.1.3 Span wrapper and naming conventions

The companion file `observability/spans.ts` defines `withSpan`, a small helper that wraps asynchronous work in a traced span and ensures that latency and error metrics are updated consistently:

```
export async function withSpan<T>(
  name: string,
  attrs: Record<string, any>,
  fn: () => Promise<T>
): Promise<T> {
  const tr = getTracer();
  const span = tr.startSpan?.(name, { attributes: attrs });
  const t0 = performance.now();
  try {
    const res = await fn();
    try { span?.setAttribute?.('ok', true); } catch {}
    span?.end?.();
    try { METRICS.reqLatencyMs.record?.(performance.now() - t0); } catch {}
    return res;
  } catch (e: any) {
    try {
      span?.setAttribute?.('ok', false);
      span?.setAttribute?.('error', e?.message ?? String(e));
    } catch {}
    span?.end?.();
    try { METRICS.errors.add?.(1); } catch {}
    throw e;
```

```
  }
}
```

Given a span name n, an attribute map A and an asynchronous function f, we can view `withSpan` as implementing the higher-order transformation

$$\text{withSpan}(n, A, f) \;=\; \begin{cases} \text{record latency } \ell, \text{ set } \texttt{ok} = 1, & \text{if } f \text{ resolves successfully,} \\ \text{record error, increment } \texttt{errors\_total}, & \text{if } f \text{ rejects or throws,} \end{cases}$$

with $\ell = t_{\text{end}} - t_{\text{start}}$ tracked via `performance.now()`. All other details (span creation, attribute setting, metric recording) are encapsulated.

In the current codebase, this helper is used primarily in the AI orchestrator (`services/agents/orchestrator.ts`), where span names follow a dotted convention such as `"orchestrator.plan"`, `"orchestrator.dryrun"`, `"orchestrator.code"`, `"orchestrator.task"`, and `"orchestrator.critic"`. This naming scheme reflects the nested structure of an AI-driven coding session: each span corresponds to a logical phase in the plan–execute–critique loop. Attributes may include the current round, model route, or task identifiers, thereby enabling fine-grained inspection in a tracing backend.

The same pattern can be extended to other subsystems:

- *Flows*: spans named `"flow.safety.run"` or `"flow.capacity.run"` can wrap complete flow executions, with attributes capturing `flowId`, patient pseudonyms, and run durations.
- *Timers*: spans named `"timer.session"` can measure elapsed time for session timers, with attributes indicating mode (stopwatch vs countdown) and number of laps.
- *Exports and tools*: spans such as `"tools.consulton.export"` can wrap document-generation operations in the Tools module, correlating latency and error rates with particular export recipes.

Because `withSpan` is agnostic to the nature of the underlying operation, these naming conventions are purely semantic and can be adapted to the needs of individual sites. What matters for clinical informatics is that each high-level activity—an AI call, a flows run, a timer-controlled session, a bulk export—can be turned into a span with a clear, stable name and a small vocabulary of attributes (e.g. patient type, risk category, environment, provider).

12.1.4 End-to-end pipeline

Figure 35 summarises the end-to-end journey of an instrumented operation within SynapseCore, from a user or AI-triggered action through to a potential OTEL backend. The design explicitly supports environments where no OTEL backend is configured (in which case the pipeline collapses to no-op tracer and meter) as well as environments where a full tracing stack is present.

By keeping the OTEL wiring thin and relying on clear naming and metric semantics, the observability subsystem provides a bridge between day-to-day clinical use of the workbench and the kind of operational analytics required by health systems: latency profiling of AI routes, early detection of error spikes, monitoring of token and cost trajectories, and, ultimately, data-driven refinement of clinical decision-support algorithms.

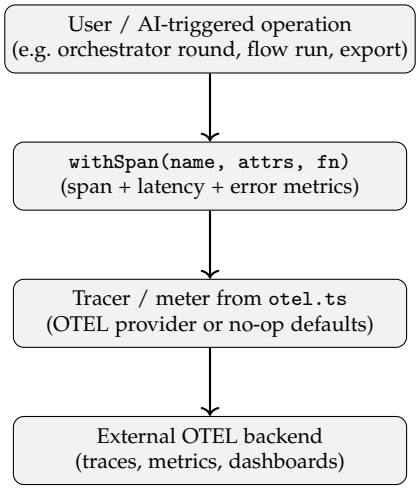


Figure 35: *End-to-end observability path for an instrumented operation. The same abstraction applies to AI calls, flows, timer-controlled sessions, and export tasks.*

12.2 High-Level Metrics

While Section 12.1 focuses on the mechanics of OpenTelemetry integration and the `METRICS` map, the present subsection formalises the high-level metrics we track and how they can be aggregated into clinically meaningful views. Conceptually, the system observes a stream of AI-related events

$$e_i = (t_i, p_i, m_i, r_i, s_i, c_i, \ell_i, P_i, C_i, k_i, \varepsilon_i),$$

where each event e_i corresponds to a single logical operation such as an AI completion, a flow-run summary, or an export. For event i:

- t_i is the timestamp.
- p_i is the provider (e.g. OpenAI, Anthropic).
- m_i is the model identifier (e.g. `gpt-4.1`).
- r_i is the route or configuration key (provider, model, sampling).
- s_i is the session identifier (a consultation or workbench session).
- c_i is the clinical task identifier (e.g. risk summary, capacity assessment, medication letter).
- ℓ_i is the latency in milliseconds.
- P_i and C_i are prompt and completion tokens, respectively.
- k_i is the estimated monetary cost in USD.
- $\varepsilon_i \in \{0, 1\}$ encodes whether the event ended in an error (1) or not (0).

The instruments in the `METRICS` object (`req_latency_ms`, `tokens_prompt`, `tokens_completion`, `cost_usd`, `errors_total`, `rate_limit_hits`, `cache_hits`) implement streaming aggregates of these quantities, typically grouped by a subset of the indices (p, m, r, s, c). This allows us to analyse performance from three complementary vantage points:

1. *Provider/model level*: behaviour of each model route across all users and tasks.
2. *Session level*: behaviour within a single clinical consultation or workbench session.
3. *Task level*: behaviour tied to specific clinical tasks, such as “Generate risk summary” or “Draft capacity letter”.

Table 56 summarises the main metrics and their aggregation levels.

Table 56: *High-level metrics tracked in SynapseCore and their typical aggregation levels. The same underlying OTEL instruments can be sliced by provider, model route, session, and clinical task.*

Metric	Aggregation level	Example clinical / operational question
Latency ℓ (ms) per provider/model	Provider / model route	"How fast is model m from provider p when generating risk summaries compared to consultation letters?"
Prompt/completion tokens (P, C)	Session / task / provider	"How many tokens are spent on a typical 50 minute consultation when using a particular AI route for MBC plus flows?"
Cost k (USD)	Provider route / institution	"What is the monthly cost of running AI-assisted documentation for all outpatient sessions at this clinic?"
Error count ε and rate-limit hits	Provider / route / time window	"Are we seeing spikes in errors or rate-limit responses when stress-testing a new provider or model?"
Cache hits	Provider / route / endpoint	"For which prompts or templates is caching most effective in reducing latency and cost?"

12.2.1 Latency per provider and model

For each provider–model pair (p, m), we consider the multiset of latencies

$$\mathcal{L}_{p,m} = \{\ell_i \mid p_i = p,\ m_i = m\}.$$

Over a time window $[0, T]$, empirical statistics such as the mean, variance, and quantiles of $\mathcal{L}_{p,m}$ are defined as usual:

$$\bar{\ell}_{p,m}(T) = \frac{1}{N_{p,m}(T)} \sum_{i:p_i=p,\ m_i=m,\ t_i \le T} \ell_i,$$

where $N_{p,m}(T)$ is the number of events with $p_i = p$, $m_i = m$ and $t_i \le T$. In practice, the histogram instrument `req_latency_ms` in `METRICS` approximates the distribution of ℓ_i and exposes quantiles (e.g. median, P95, P99) via an OTEL backend.

Clinically, latency at the provider/model level is not just a matter of engineering convenience. A route with excellent narrative quality but prohibitive latency may be unsuitable for real-time bedside usage. Conversely, a slightly weaker model that reliably returns within a few hundred milliseconds might be preferable for interactive flows (for example, iterative edits to a risk summary while the patient is still present). By stratifying $\mathcal{L}_{p,m}$ by task c (e.g. risk summary vs. medication letter), informatics teams can select routes that match the temporal affordances of different parts of the consultation.

12.2.2 Token usage per session and clinical task

Token counts are recorded via the counters `tokens_prompt` and `tokens_completion`, which increment by P_i and C_i at each event. For a fixed session s (e.g. a single outpatient visit) we define the total token usage

$$\mathrm{Tok}_s^{\mathrm{prompt}} = \sum_{i:s_i=s} P_i, \qquad \mathrm{Tok}_s^{\mathrm{compl}} = \sum_{i:s_i=s} C_i.$$

These quantities can be interpreted as a measure of how heavily the AI workbench was used during that session. For instance, a follow-up visit focusing on MBC review may have

modest token usage, whereas an initial diagnostic consultation with extensive flows, letters, and psychoeducation may exhibit substantially larger values.

Similarly, for each clinical task c (such as “MSE summary”, “Capacity assessment letter”, or “Group programme summary”) we can define task-level aggregates

$$\mathrm{Tok}_c^{\mathrm{prompt}} = \sum_{i : c_i = c} P_i, \qquad \mathrm{Tok}_c^{\mathrm{compl}} = \sum_{i : c_i = c} C_i,$$

optionally stratified further by provider and model. Such breakdowns support questions of the form:

- “Which tasks are driving the majority of token usage (and therefore cost)?”
- “Can we move less critical tasks to cheaper models without compromising quality?”
- “How does token usage per session vary between services (e.g. CAMHS vs. adult acute)?”

Because token counts are closely correlated with provider charges, the same aggregates can be reweighted by per-token prices to form session- and task-level cost estimates. If $c_{p,m}^{\mathrm{prompt}}$ and $c_{p,m}^{\mathrm{compl}}$ denote the per-token prices for provider p and model m, the total cost for a session s can be approximated by

$$\mathrm{Cost}_s \approx \sum_{i : s_i = s} \left(P_i \, c_{p_i, m_i}^{\mathrm{prompt}} + C_i \, c_{p_i, m_i}^{\mathrm{compl}} \right),$$

which is also tracked by the `cost_usd` counter.

12.2.3 Error and rate-limit events

Reliability is captured through error and rate-limit metrics. The `errors_total` counter increments whenever a traced operation wrapped in `withSpan` throws or rejects, while `rate_limit_hits` counts responses identified as rate-limit events by the provider client or orchestration layer.

For a given provider/model pair (p, m) and time window $[0, T]$, the empirical error rate is

$$\mathrm{ErrRate}_{p,m}(T) = \frac{\sum_{i : p_i = p,\ m_i = m,\ t_i \le T} \varepsilon_i}{N_{p,m}(T)},$$

with $\varepsilon_i = 1$ for errors and 0 otherwise. A similar definition applies to rate-limit events if we replace ε_i with a rate-limit indicator ρ_i and $N_{p,m}(T)$ with the number of calls that could plausibly trigger rate limiting.

In clinical terms, elevated error or rate-limit rates can manifest as interrupted workflows (e.g. AI summaries failing midway through a consultation) or forced fallbacks to generic routes. By tracking these rates over time and correlating them with configuration changes (e.g. enabling a new provider, adjusting sampling), informatics teams can detect regressions early and enforce safe defaults.

12.2.4 Aggregation pipeline

High-level metrics are obtained by aggregating event-level quantities through a small number of stages, as illustrated in Figure 36. At the lowest level, every instrumented operation (AI call, flow export, timer-controlled audit step) yields an event e_i. These events feed OTEL instruments via `METRICS`, which in turn expose aggregated views to dashboards or downstream analytics.

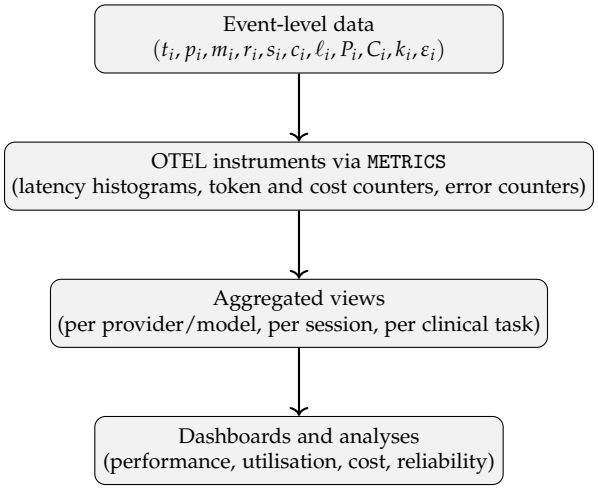


Figure 36: *Aggregation pipeline from event-level quantities to high-level metrics. The same small set of OTEL instruments supports multiple perspectives: provider/model performance, session-level usage, and task-level cost and reliability.*

By aligning latency, token, cost, and error metrics along clinically meaningful dimensions (session, task, service line), SynapseCore enables a form of *operational measurement-based care*: not only are patient outcomes and symptoms quantified, but the behaviour of the digital workbench itself is rendered measurable and optimisable.

12.3 AI Route Telemetry

The Enhanced IDE allows users to switch flexibly between different AI "routes"—combinations of provider (e.g. OpenAI, Anthropic, Gemini, Ollama) and model (e.g. `gpt-4.1`, `claude-3.5`). Because route changes affect both clinical behaviour (style of summaries, hallucination profile) and operational characteristics (latency, cost), they are treated as first-class telemetry events.

The module `observability/aiRouteTelemetry.ts` provides a thin, browser-side bridge between route changes in the IDE, the global telemetry queue in `components/ai/telemetry/events.ts`, and the toast notification system in `ui/toast/api`. It implements three responsibilities:

1. Debounced detection of route changes to avoid flooding the telemetry system when the user experiments with multiple routes in quick succession.
2. Emission of a structured telemetry event of type `"ai_route_changed"` describing the previous and current route.
3. Presentation of a concise toast notification so that the clinician is always aware of the active AI route.

12.3.1 Debounced detection of route changes

At module scope, the file declares a small amount of state:

```
let pending: {
  prev: { provider: string; model: string };
  curr: { provider: string; model: string };
} | null = null;
```

```
let timer: number | null = null;
const DEBOUNCE_MS = 300;
```

The exported function `emitAiRouteChanged(previous, current)` is called whenever the AI configuration store observes a change in the selected provider or model. Rather than emitting telemetry immediately, the function stores the most recent pair in `pending` and (re)starts a short timeout:

```
export function emitAiRouteChanged(previous, current) {
  pending = { prev: previous, curr: current };
  if (timer) { clearTimeout(timer); }
  timer = window.setTimeout(() => { ... }, DEBOUNCE_MS);
}
```

Within the timeout handler, if `pending` is still non-null, the change is considered stable and a telemetry event and toast are emitted. If, however, the user changes the route again within `DEBOUNCE_MS` milliseconds, the timeout is cancelled and a new one is scheduled. This is the standard debouncing pattern.

Formally, let $\{(t_k, r_k)\}_{k=1}^{\infty}$ denote the sequence of route selections, where t_k is the wall-clock time of the k-th selection and $r_k = (p_k, m_k)$ encodes the provider and model. For a fixed debounce interval $\Delta > 0$, we define an emitted route-change event whenever a gap of at least Δ occurs after a selection:

$$E_k = \begin{cases} 1, & \text{if } t_{k+1} - t_k \geq \Delta, \\ 0, & \text{otherwise,} \end{cases}$$

and the associated effective change is from r_k to r_{k+1}. In `aiRouteTelemetry.ts`, Δ is instantiated as `DEBOUNCE_MS = 300`, meaning that if the user briefly scrubs through several provider/model combinations in under 300 ms, only the final stabilised choice produces a telemetry event.

This debouncing has two important consequences for clinical informatics:

- *Telemetry stability*: downstream analyses of route usage (Section 12.2) reflect committed choices rather than transient UI exploration.
- *Reduced noise*: tracing and metrics backends are not flooded with route-change events when users are experimenting with different configurations.

12.3.2 Telemetry event emission

Once the debounce interval has elapsed without further changes, the timeout callback reads the stored pair `pending.prev` and `pending.curr` and emits a structured telemetry event via `telemetryEmit`:

```
const { prev, curr } = pending;
telemetryEmit({
  type: 'ai_route_changed',
  previous: prev,
  current: curr,
} as any);
```

Table 57: Schema for the "ai_route_changed" telemetry event emitted by aiRouteTelemetry.ts.

Field	Type	Interpretation
type	String (literal)	Event type, always "ai_route_changed".
previous.provider	String identifier	Provider key before the change (e.g. "openai", "anthropic", "gemini", "ollama").
previous.model	String identifier	Model key before the change (e.g. "gpt-4.1", "claude-3.5-sonnet").
current.provider	String identifier	Provider key after the change.
current.model	String identifier	Model key after the change.

Here, telemetryEmit is the emit function exported from components/ai/telemetry/events.ts. It appends the event object to a global in-memory queue on window:

```
export type TelemetryEvent = { type: string; [k: string]: unknown };

export function emit(evt: TelemetryEvent) {
  try {
    (window as any).telemetry = (window as any).telemetry || {};
    const q: TelemetryEvent[] =
      ((window as any).telemetry.queue ||= []);
    q.push(evt);
    if (process.env.NODE_ENV !== 'production') {
      console.log('[telemetry]', evt);
    }
  } catch {}
}
```

An external host (for example, a VS Code extension or a small background worker) can periodically drain this queue and forward events to a central OTEL collector or logging system, where they can be correlated with other metrics such as latency and cost.

The "ai_route_changed" event has the schema shown in Table 57.

At analysis time, these events can be joined with the high-level metrics from Section 12.2 to reconstruct per-route trajectories (e.g. how often a particular clinician switches from a fast but shallow model to a slower but more nuanced one) or to understand institution-wide adoption of different provider/model combinations.

12.3.3 Toast notifications and user-facing feedback

In addition to emitting telemetry, the same timeout callback presents a small toast notification to the user, using the showToast helper:

```
try {
  const label = `${friendlyProvider(curr.provider)}  ${curr.model}`;
  showToast({
    kind: 'info',
    message: `AI route: ${label}`,
    contextKey: 'ai:route-change',
  });
} catch {}
```

The helper function `friendlyProvider` maps internal provider keys to human-readable names:

```
function friendlyProvider(p: string): string {
  if (p === 'google' || p === 'gemini') return 'Gemini';
  if (p === 'openai') return 'OpenAI';
  if (p === 'anthropic') return 'Anthropic';
  if (p === 'ollama') return 'Ollama';
  return p;
}
```

Thus, if the route is changed from `previous = { provider: 'openai', model: 'gpt-4.1' }` to `current = { provider: 'anthropic', model: 'claude-3.5' }`, the resulting toast might read:

"AI route: Anthropic • claude-3.5".

The `contextKey "ai:route-change"` ensures that repeated route-change toasts are deduplicated and styled consistently within the global toast system.

From a clinical workflow perspective, this explicit feedback serves as a form of *cognitive guardrail*. A common concern in AI-assisted practice is that clinicians may inadvertently assume that a particular provider or model is active while a different one is actually serving requests. By showing the route label after each stabilised change, the interface externalises this information and reduces the risk of misinterpretation (for example, expecting a safety-tuned model when a general-purpose model is configured).

12.3.4 End-to-end route telemetry flow

Figure 37 combines these elements into a single flow. At the top, a user interacts with the AI configuration controls inside the Enhanced IDE. Route changes are observed and debounced by `emitAiRouteChanged`. Once stabilised, they produce a telemetry event and a toast notification. The event is queued on `window.telemetry.queue` and can be exported to an OTEL backend, while the toast provides immediate, user-facing feedback.

By combining debounced event emission with explicit user feedback, SynapseCore links the abstract notion of "AI route" to both observable telemetry and clinician-facing cues. This makes provider and model selection auditable at scale and simultaneously transparent to individual users, which is essential when AI-assisted reasoning is embedded in high-stakes psychiatric workflows.

12.4 Future Analytics

The observability primitives described in Sections 12.1–12.3 are intentionally minimal: they expose event-level quantities (latency, tokens, cost, errors) and structured route-change events without prescribing how these data should be analysed. The goal of this subsection is to outline a forward-looking analytics layer built on top of those primitives. This layer is intended to support data-driven decisions about model choice, cost/performance trade-offs, and clinical adoption, and to provide dashboards that integrate seamlessly with the Enhanced IDE and psychiatry modules.

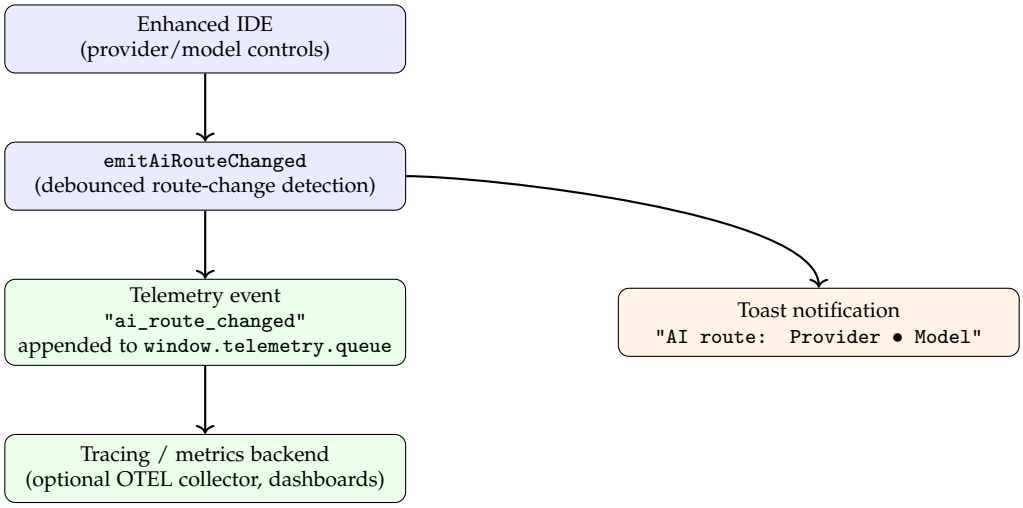


Figure 37: *AI route telemetry pipeline. Provider/model changes in the Enhanced IDE are debounced, then emitted as structured telemetry events and shown to the user as toast notifications. Events can be exported to an OTEL backend, while toasts provide immediate contextual feedback at the point of care.*

12.4.1 Aggregating metrics into route-level profiles

Let $\mathcal{R}$ denote the finite set of AI routes available in the system, where each route $r \in \mathcal{R}$ corresponds to a specific provider/model configuration (and possibly temperature or other sampling parameters). From Section 12.2, we obtain for each event i an associated route r_i and metrics $(\ell_i, P_i, C_i, k_i, \varepsilon_i)$.

For a fixed route r and time window $[0, T]$, we define route-level aggregates:

$$\bar{\ell}_r(T) = \frac{1}{N_r(T)} \sum_{i: r_i = r,\, t_i \le T} \ell_i, \qquad \bar{k}_r(T) = \frac{1}{N_r(T)} \sum_{i: r_i = r,\, t_i \le T} k_i,$$

$$\mathrm{ErrRate}_r(T) = \frac{1}{N_r(T)} \sum_{i: r_i = r,\, t_i \le T} \varepsilon_i,$$

where $N_r(T)$ is the number of events routed through r in the window. In addition, we compute route-level token averages

$$\bar{P}_r(T) = \frac{1}{N_r(T)} \sum_{i: r_i = r,\, t_i \le T} P_i, \qquad \bar{C}_r(T) = \frac{1}{N_r(T)} \sum_{i: r_i = r,\, t_i \le T} C_i.$$

These aggregates can be stratified further by clinical task c (for example, risk summary, MSE narrative, capacity letter) and by service line (e.g. adult acute vs. perinatal vs. CAMHS). The basic building block is therefore a route–task pair (r, c) with a small vector of summary statistics

$$\mathbf{m}_{r,c}(T) = \big(\bar{\ell}_{r,c}(T),\, \bar{k}_{r,c}(T),\, \mathrm{ErrRate}_{r,c}(T),\, \bar{P}_{r,c}(T),\, \bar{C}_{r,c}(T)\big),$$

computed over sliding windows such as the last 7 days, 30 days, or quarter. In practice, these profiles would be materialised in a warehouse or time-series database fed by OTEL exporters.

Table 58 sketches the intended content of a "route profile" view.

12.4.2 Multi-objective route optimisation

Given route profiles, the next step is to define an optimisation problem that captures clinically relevant trade-offs. For many tasks, latency, cost, and reliability pull in different directions;

Table 58: *Illustrative fields in a route-level profile used for optimisation and monitoring. Values are typically aggregated over recent time windows and stratified by clinical task.*

Field	Example	Interpretation
Route identifier r	`openai:gpt-4.1`	Provider and model (and optionally temperature or other settings) defining a unique AI route.
Mean latency $\bar{\ell}_{r,c}$	450 ms (risk summaries)	Typical end-to-end time for a given task c on route r.
Mean cost $\bar{k}_{r,c}$	\$0.03 per call	Average spend per invocation of task c on route r.
Error rate $\mathrm{ErrRate}_{r,c}$	0.4%	Fraction of invocations with errors (network, provider, safety filters, etc.).
Token usage $(\bar{P}_{r,c}, \bar{C}_{r,c})$	(900, 350) tokens	Typical prompt and completion sizes for task c on r.

no single route dominates all others. We therefore adopt a multi-objective view.

For each route r and task c, consider the vector

$$\mathbf{z}_{r,c} = \left(\bar{\ell}_{r,c},\, \bar{k}_{r,c},\, \mathrm{ErrRate}_{r,c}\right) \in \mathbb{R}^3_{\geq 0}.$$

A route r is *Pareto dominated* by another route r' (for the same task c) if

$$\bar{\ell}_{r',c} \leq \bar{\ell}_{r,c}, \quad \bar{k}_{r',c} \leq \bar{k}_{r,c}, \quad \mathrm{ErrRate}_{r',c} \leq \mathrm{ErrRate}_{r,c},$$

with at least one strict inequality. The set of *Pareto-efficient* routes $\mathcal{R}^\star_c$ for task c is defined as those routes that are not dominated by any other. Informally, these are the routes for which no other configuration is better on all three metrics simultaneously.

In practice, decision-makers often wish to collapse the three objectives into a scalar "score" in order to define default routes. One natural approach is to normalise each metric and form a weighted sum

$$J_{r,c} = \alpha_c\, \tilde{\ell}_{r,c} + \beta_c\, \tilde{k}_{r,c} + \gamma_c\, \widetilde{\mathrm{ErrRate}}_{r,c},$$

where the tildes denote normalised quantities (for example, min-max scaling over $\mathcal{R}$) and $(\alpha_c, \beta_c, \gamma_c)$ are task-specific weights reflecting clinical preferences:

- For time-critical tasks (e.g. safety summaries during agitation), we might choose $\alpha_c \gg \beta_c, \gamma_c$.
- For low-stakes, high-volume tasks (e.g. drafting teaching material), we might emphasise cost (β_c) while tolerating slightly higher latency.
- For medico-legal documents (e.g. capacity letters), we may prioritise low error rates (γ_c) even at the expense of latency and cost.

The *recommended* route for task c would then be

$$r^\star_c = \arg\min_{r \in \mathcal{R}^\star_c} J_{r,c},$$

i.e. the lowest-weighted score among Pareto-efficient routes. This decision rule remains transparent: weights are explicit and can be reviewed by governance panels, and changes in the underlying metrics (e.g. a provider deploying a faster model) can be propagated automatically.

12.4.3 Dashboard integration and feedback loops

The final piece is to present these aggregates and optimisation results in dashboards that are accessible both to informatics teams and to clinicians with an interest in service quality. A

Table 59: *Illustrative dashboard tiles for future analytics. Each tile corresponds to a visualisation that can be built directly from the metrics and route-change events described in Sections 12.2–12.3.*

Dashboard tile	Primary slice	Key question
Latency heatmap	Provider × task	"Which routes are fast enough for interactive use, and which are better suited to offline summarisation?"
Cost per 100 sessions	Service line (e.g. adult, CAMHS)	"How much are we spending on AI support for each service line and task type?"
Error and rate-limit trends	Provider / route / time	"Are we seeing instability in any provider or model that might warrant a temporary fallback route?"
Route adoption over time	Route × week	"How quickly are new models being adopted across clinicians and teams?"

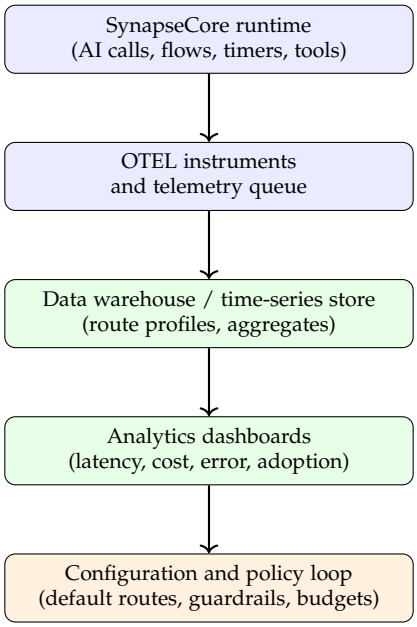


Figure 38: *Conceptual future analytics flow. Event-level telemetry from SynapseCore is exported via OTEL, aggregated into route and task-level profiles, visualised on dashboards, and then fed back into configuration and governance decisions.*

future analytics stack might include:

- A streaming pipeline from OTEL exporters to a central data warehouse or time-series database.
- A set of pre-defined, psychiatry-specific dashboards (for example, created in Grafana, Metabase, or a similar tool) that visualise route-level metrics, cost trajectories, and error patterns.
- Tight integration with the Enhanced IDE and psychiatry modules, so that recommended routes and alerts are surfaced contextually.

Table 59 lists exemplar dashboard tiles and the questions they address.

Figure 38 summarises the envisioned end-to-end analytics flow, connecting SynapseCore telemetry to dashboards and then back to configuration decisions.

12.4.4 Clinical governance and research opportunities

From a psychiatric perspective, such an analytics layer serves several roles beyond infrastructure optimisation:

- *Governance*: committees responsible for digital safety can review empirical latency, cost, and error data before approving new routes for clinical use, and can monitor whether usage patterns align with expectations (for example, ensuring that high-risk tasks use conservative, safety-tuned models).
- *Quality improvement*: teams can run time-limited experiments (A/B tests) comparing different routes for specific tasks and assess both operational metrics and clinical outcomes (symptom change, admission rates, re-attendance).
- *Research*: de-identified, aggregate telemetry can support studies of how clinicians interact with AI tools over time, which tasks benefit most from assistance, and how AI usage correlates with patient-level outcomes in future deployments.

In this sense, future analytics are not an afterthought but an integral component of measurement-based care: the same philosophy that drives systematic outcome measurement at the patient level is applied to the AI infrastructure itself, enabling iterative, evidence-informed refinement of the digital psychiatry workbench.

Chapter III: Formalisation and Learning Components

Sections 13–14

Overview. Part III provides a mathematical and algorithmic treatment of SynapseCore. Section 13 introduces formal models that capture the system as a typed graph, represent session timelines, and characterise flows as finite state machines. For example, the clinical workspace at time t is represented as a labelled graph

$$G_t = (V_t, E_t, \tau_t),$$

where V_t are nodes (sections, cards, timers, AI calls), E_t are edges (containment, dependency, temporal order), and τ_t is a typing function assigning semantic categories to nodes and edges. Section 14 then discusses session-level machine learning modules, including feature engineering for session patterns, TensorFlow.js model design, and data pipelines for prospective analytics.

At a global level, Part III formalises the transformation

$$(K_{\text{psy}}, F_{\text{flows}}, T_{\text{timer}}) \longmapsto (G_t, g_\phi),$$

where G_t provides a structured semantic view of system state and g_ϕ denotes the learned models over sessions and flows. This formal layer underpins future analytical and safety-oriented work while keeping the architecture in Part II grounded in explicit mathematical objects.

13 Formal Models and Mathematical Foundations

13.1 System as a Typed Graph

To reason about the correctness and composability of SynapseCore at a global level, it is useful to model the system as a *typed directed graph*. Nodes correspond to major modules (AI panel, psychiatry, flows, timer, tools, IDE shell, and shared stores), while edges encode data flows between these modules (context objects, prompts, exports, configuration updates).

13.1.1 Typed directed graph structure

Let

$$G = (V, E)$$

be a directed graph, where V is the set of nodes (modules) and $E \subseteq V \times V$ is the set of directed edges. We equip G with two typing functions:

$$\tau_V : V \to T, \qquad \tau_E : E \to L,$$

where T is the finite set of *module types* (for example, *UI shell*, *clinical module*, *AI coordination*, *store*) and L is the set of *edge labels* describing payload types (for example, *context*, *prompt*, *export*, *configuration*).

We focus on the following distinguished nodes:

$$V = \{v_{\text{IDE}}, v_{\text{AI}}, v_{\text{psych}}, v_{\text{flows}}, v_{\text{timer}}, v_{\text{tools}}, v_{\text{stores}}\}.$$

Informally:

- v_{IDE} represents the Enhanced IDE shell and layout (center panel shell, sidebars, theme and key bindings).
- v_{AI} represents the AI panel and orchestration layer (model registry, sampling mapper, route selection).
- v_{psych} represents the psychiatry module (content cards, MBC engine, clinical prompts).
- v_{flows} represents structured clinical flows (agitation, safety, capacity, catatonia, observation, etc.).
- v_{timer} represents the session-timer subsystem (stopwatch, countdown, laps).
- v_{tools} represents the Tools module (exports, audit, snapshots, de-identification).
- v_{stores} collects state stores (Zustand-like stores for AI config, flows, psychiatry, timer, tools, theme).

The node-typing function τ_V can then be specified as:

$$\begin{aligned}
\tau_V(v_{\text{IDE}}) &= \text{UI shell},\\
\tau_V(v_{\text{AI}}) &= \text{AI coordination},\\
\tau_V(v_{\text{psych}}) &= \text{clinical module},\\
\tau_V(v_{\text{flows}}) &= \text{clinical module},\\
\tau_V(v_{\text{timer}}) &= \text{utility module},\\
\tau_V(v_{\text{tools}}) &= \text{clinical tooling},\\
\tau_V(v_{\text{stores}}) &= \text{state store}.
\end{aligned}$$

On the edge side, we distinguish the following payload labels:

$$\begin{aligned}
L = \{&\text{ContextPayload},\ \text{PromptPayload},\ \text{ExportPayload},\\
&\text{ConfigUpdate},\ \text{ViewState}\}.
\end{aligned}$$

Table 60: *Nodes in the system-level typed graph and their associated state spaces. Names correspond to high-level modules rather than individual files.*

Node	Type $\tau_V(v)$	State space $\Sigma(v)$ (schematic)
v_{IDE} (IDE shell)	UI shell	Layout, theme, keybindings, active center panel and sidebars.
v_{AI} (AI panel)	AI coordination	AI route config, in-flight requests, transcripts, streaming responses.
v_{psych} (psychiatry)	Clinical module	Card selections, modal open state, MBC configurations, prompts.
v_{flows} (flows)	Clinical module	Flow run states (per flow), library selection, outcome objects.
v_{timer} (timer)	Utility module	Timer mode, elapsed time, remaining time, laps.
v_{tools} (tools)	Clinical tooling	Export recipes, selected templates, snapshot selections, de-identification flags.
v_{stores} (stores)	State store	Product of all global stores (configuration and view-state).

We write $\tau_E(u \to v) \in L$ for the label of an edge from u to v. For example, psychiatry and flows send *context* to the AI panel, the AI panel sends *exportable* artefacts to tools, and stores send *configuration* and *view-state* to all other modules.

13.1.2 Node taxonomy and state spaces

Each node $v \in V$ maintains a local state space $\Sigma(v)$, which corresponds to the TypeScript store or React component state underlying the module. For instance:

$$\Sigma(v_{\text{AI}}) = \text{AiConfigState} \times \text{AiPanelState},$$

where `AiConfigState` includes provider, model, temperature, and safety flags, and `AiPanelState` includes transcript snippets, current prompt, and streaming output. Similarly:

$$\Sigma(v_{\text{psych}}) = \text{PsychiatryModalState} \times \text{ContentSelection},$$
$$\Sigma(v_{\text{flows}}) = \text{FlowRunState} \times \text{FlowLibraryState},$$
$$\Sigma(v_{\text{timer}}) = \text{TimerState}, \qquad \Sigma(v_{\text{tools}}) = \text{ToolsState},$$
$$\Sigma(v_{\text{IDE}}) = \text{LayoutState} \times \text{ThemeState} \times \text{KeybindingState},$$
$$\Sigma(v_{\text{stores}}) = \prod_j \text{Store}_j,$$

where the product ranges over all registered global stores (AI config store, flows UI store, psychiatry store, timer store, tools store, theme store, etc.).

Table 60 summarises the node taxonomy and their typical state spaces.

13.1.3 Edge taxonomy and data flows

Edges in E capture the typed data flows between modules. At a coarse level, we identify the following classes of edges:

1. *Context edges*: domain modules to AI panel.

   $$E_{\text{ctx}} = \{v_{\text{psych}} \to v_{\text{AI}}, v_{\text{flows}} \to v_{\text{AI}}, v_{\text{timer}} \to v_{\text{AI}}\}, \quad \tau_E(e) = \text{ContextPayload } \forall e \in E_{\text{ctx}}.$$

   These edges represent the construction of structured context objects (for example, MSE snippets, risk notes, flow outcomes, session timing) that feed the AI prompt builder.

2. *Prompt and result edges*: AI panel to tools and IDE.

$$E_{\text{ai}} = \{v_{\text{AI}} \to v_{\text{tools}}, v_{\text{AI}} \to v_{\text{IDE}}\}, \quad \tau_E(v_{\text{AI}} \to v_{\text{tools}}) = \text{ExportPayload},$$

   where `ExportPayload` includes formatted notes, letters, and JSON exports; the edge to v_{IDE} carries `ViewState` updates (for example, to show AI drafts in the center panel).
3. *Configuration edges*: stores to all modules.

$$E_{\text{cfg}} = \{v_{\text{stores}} \to v \mid v \in V \setminus \{v_{\text{stores}}\}\}, \quad \tau_E(e) = \text{ConfigUpdate}\ \forall e \in E_{\text{cfg}}.$$

   These edges encode subscription relationships to global stores (for example, the AI panel subscribing to AI config changes, flows subscribing to theme changes).
4. *View-state edges*: IDE to modules.

$$E_{\text{view}} = \{v_{\text{IDE}} \to v \mid v \in \{v_{\text{psych}}, v_{\text{flows}}, v_{\text{timer}}, v_{\text{tools}}, v_{\text{AI}}\}\}, \quad \tau_E(e) = \text{ViewState}\ \forall e \in E_{\text{view}}.$$

   These edges correspond to layout decisions (which module is shown where) and selection events (which flow or card is active).

In total, the edge set is

$$E = E_{\text{ctx}} \cup E_{\text{ai}} \cup E_{\text{cfg}} \cup E_{\text{view}},$$

with each edge explicitly typed by τ_E. This structure captures the major pathways by which information flows through the system without requiring us to enumerate every individual React property or store subscription.

13.1.4 Composition along paths

The main benefit of this formalisation is that it allows us to reason about *compositions* along paths in G. A directed path

$$\pi = (v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} \dots \xrightarrow{e_n} v_n)$$

is *type-safe* if the payload and state types along the path compose appropriately. For each edge $e_k : v_{k-1} \to v_k$, we can view the underlying implementation as a function

$$f_{e_k} : \Sigma(v_{k-1}) \to \Sigma(v_k) \times \mathcal{Y}_{e_k},$$

where $\mathcal{Y}_{e_k}$ is the payload space associated with $\tau_E(e_k)$. For example, a context edge from psychiatry to AI panel might implement a mapping

$$f_{v_{\text{psych}} \to v_{\text{AI}}} : \Sigma(v_{\text{psych}}) \to \Sigma(v_{\text{AI}}) \times \text{ContextPayload}.$$

Given a path π, the overall transformation is the composition

$$F_\pi = f_{e_n} \circ f_{e_{n-1}} \circ \cdots \circ f_{e_1},$$

which is well-defined if and only if the intermediate types match. Concretely, an end-to-end path representing "psychiatry card to AI summary to tools export" might be

$$\pi_{\text{summary}} = \big(v_{\text{psych}} \to v_{\text{AI}} \to v_{\text{tools}}\big),$$

with composition

$$F_{\pi_{\text{summary}}} : \Sigma(v_{\text{psych}}) \to \Sigma(v_{\text{tools}}) \times \text{ExportPayload},$$

where the intermediate AI panel state $\Sigma(v_{\text{AI}})$ remains internal but participates in the type-checking of the composition.

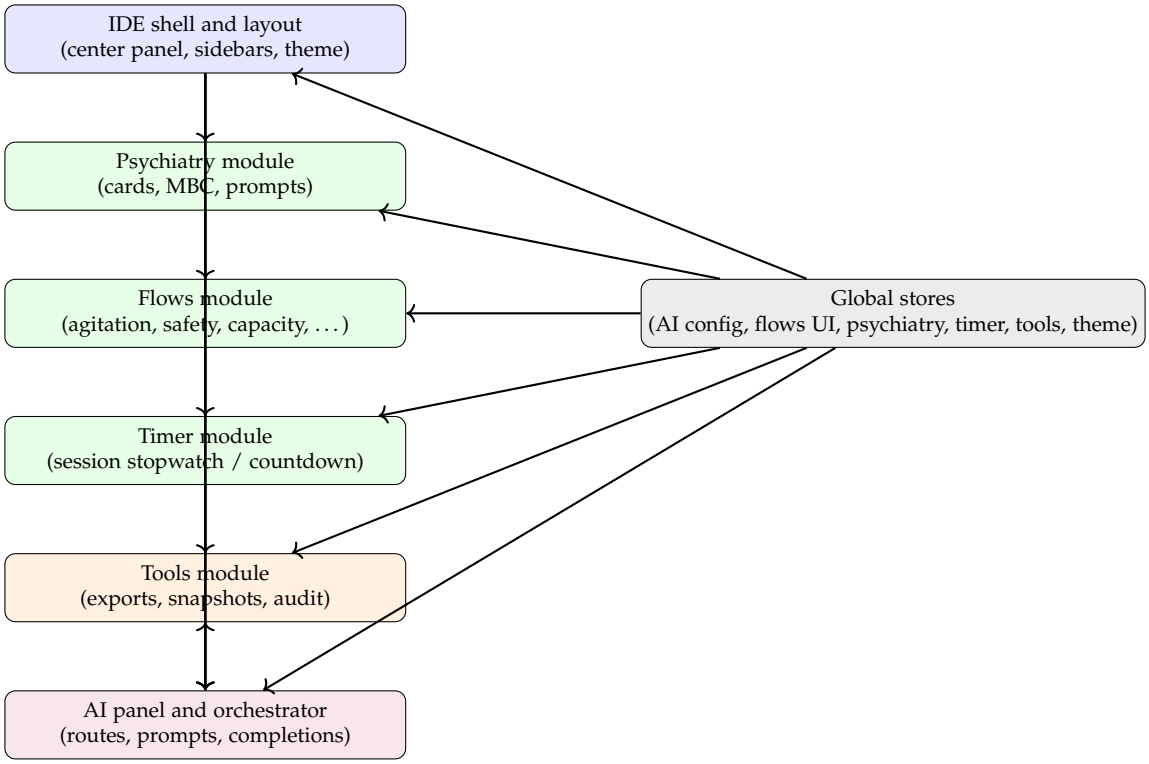


Figure 39: *Coloured, vertically oriented schema of the system-level typed graph. The IDE shell orchestrates the visibility and selection of clinical modules, which feed typed context into the AI panel. The AI panel returns structured outputs to the Tools module, while global stores provide configuration and view-state to all modules.*

13.1.5 Graph schema and visual overview

Figure 39 shows a coloured, vertically oriented schema of the main nodes and edges of G. UI, clinical, AI, and store nodes are visually distinguished to emphasise the separation of concerns.

13.1.6 Interpretation for clinical informatics

Viewed as a typed graph, SynapseCore becomes amenable to familiar static-analysis questions:

- *Reachability*: for a given node (for example, a flows outcome object), which paths lead to an exportable document? This corresponds to checking reachability of v_{tools} from v_{flows} along type-safe paths.
- *Cut sets*: which edges must be removed (for example, by configuration) to disable all flows from a module to the AI panel? This corresponds to identifying cut sets separating v_{psych} or v_{flows} from v_{AI}.
- *Invariants*: which properties of payloads are preserved along paths (for example, de-identification guarantees from psychiatry cards through AI prompts to exports)? These can be expressed as invariants on $\mathcal{Y}_{e_k}$ for edges on the relevant paths.

This graph-theoretic perspective provides the mathematical foundation for later sections on safety properties, de-identification guarantees, and formal verification of workflow constraints, while remaining faithful to the module structure and data flows observed in the actual TypeScript implementation.

13.2 Session Timeline Model

The timer and flows subsystems together provide a natural basis for a formal model of the clinical session as a time-indexed sequence of events. This model is useful both for AI-assisted narrative generation (e.g. temporal structuring of notes) and for downstream analytics (e.g. distribution of interventions across a session).

13.2.1 Event-level representation of a session

We model a single clinical session as a finite sequence of time-stamped events

$$\mathcal{T} = \{(t_i, e_i)\}_{i=1}^{N},$$

where $0 \leq t_1 < t_2 < \cdots < t_N$ are monotonically increasing timestamps in milliseconds (or seconds) from the start of the session, and e_i is an event drawn from a finite event alphabet $\mathcal{E}$.

In the context of SynapseCore, $\mathcal{E}$ includes at least:

- *Lap events* generated by the timer subsystem when the clinician records a lap (e.g. start of risk assessment, start of medication discussion).
- *Flow events* corresponding to the start and completion of structured flows (e.g. agitation, safety, capacity).
- *Intervention events* indicating specific clinical actions (e.g. de-escalation step, medication offered).
- *Documentation events* marking AI-assisted or manual documentation segments.

Formally, we can write the event alphabet as a disjoint union

$$\mathcal{E} = \mathcal{E}_{\text{lap}} \uplus \mathcal{E}_{\text{flow}} \uplus \mathcal{E}_{\text{interv}} \uplus \mathcal{E}_{\text{doc}},$$

where each component set is itself structured. For example, a lap event has the form

$$e_i = \text{Lap}(\ell_i), \qquad \ell_i \in \text{LapLabel},$$

with `LapLabel` corresponding to the optional `label` field in the TypeScript `Lap` interface. A flow-completion event might have the form

$$e_i = \text{FlowDone}(f_i, o_i),$$

where f_i is a flow identifier (e.g. `"agitation"`, `"capacity"`) and o_i is the associated outcome object constructed by the relevant builder (for example, `agitationOutcome.ts`).

Table 61 summarises the main event types used in the session timeline model.

In implementation terms, these events can be materialised as TypeScript discriminated unions, with each variant mapped to the appropriate source (timer engine, flows host, tools exports).

13.2.2 Piecewise segmentation of the session

The event sequence $\mathcal{T}$ induces a piecewise segmentation of the session into labelled intervals. Let T_{end} denote the total session duration as reported by the timer (for example, the elapsed time when the clinician stops the session timer). We define a sequence of intervals

$$I_0 = [0, t_1), \quad I_i = [t_i, t_{i+1}) \ (1 \leq i \leq N-1), \quad I_N = [t_N, T_{\text{end}}],$$

Table 61: *Illustrative event types in the session timeline $\mathcal{T}$. Payloads are schematic and correspond to the underlying TypeScript structures used by the timer, flows, and AI subsystems.*

Event kind	Symbolic form	Payload example
Lap event	$\mathrm{Lap}(\ell)$	Labelled timestamp such as `"Risk assessment started"`, `"Family member joined"`.
Flow start	$\mathrm{FlowStart}(f)$	Flow identifier (e.g. `"agitation"`) and optional metadata (e.g. context key).
Flow completion	$\mathrm{FlowDone}(f, o)$	Flow identifier and outcome object with risk grading, interventions, and follow-up plans.
Intervention event	$\mathrm{Interv}(k, m)$	Kind k (e.g. de-escalation step, PRN medication) and a small metadata object m (dosage, response).
Documentation event	$\mathrm{Doc}(d)$	Segment of documentation activity (e.g. AI-assisted risk summary, capacity letter draft) with a reference to the underlying artefact.

so that the session $[0, T_{\text{end}}]$ is partitioned into $N+1$ contiguous segments.

To each interval I_i we attach a label

$$L_i = \lambda(I_i, \mathcal{T}, \Sigma),$$

where λ is a labelling function that may depend not only on the bounding events e_i and e_{i+1} but also on relevant state snapshots Σ (for example, which flows are currently active, which cards are open, what the timer mode is). Typical label sets might include:

- *Session phases*, such as "initial rapport", "MSE and history", "risk formulation", "interventions", "documentation".
- *Intervention types*, marking intervals dominated by a specific flow (e.g. agitation de-escalation) or by medication titration.
- *Documentation windows*, intervals predominantly used for AI-assisted note drafting or export activities.

At the simplest level, one can define λ based on the event types alone. For example:

$$L_i = \begin{cases} \text{"Flow } f \text{ in progress"}, & \text{if } e_i = \mathrm{FlowStart}(f) \text{ and no } \mathrm{FlowDone}(f, \cdot) \text{ occurs in } I_i, \\ \text{"Between laps"}, & \text{if } e_i = \mathrm{Lap}(\cdot) \text{ and } e_{i+1} = \mathrm{Lap}(\cdot), \\ \text{"Unlabelled"}, & \text{otherwise.} \end{cases}$$

More sophisticated labellings can incorporate flow outcome metadata (e.g. risk level) or timer mode (for example, a countdown used for a structured exercise vs. a free-running stopwatch).

The end result is a *piecewise-constant* session descriptor:

$$[0, T_{\text{end}}] = \bigcup_{i=0}^{N} I_i, \qquad \text{with label } L_i \text{ attached to each } I_i.$$

This representation underpins both human-readable narratives (e.g. "In the first 10 minutes, the focus was on history; the next 15 minutes were spent in a structured agitation flow; the final 10 minutes on documentation") and machine-readable analytics.

13.2.3 Instantiation from timer engine and flows

The timer engine (file `timerEngine.ts`) and the flows subsystem (`centerpanel/Flows/*`) jointly provide all the ingredients to instantiate $\mathcal{T}$.

Timer events. The timer state `TimerState` encapsulates the current elapsed time and lap list. In stopwatch mode, the elapsed time t is updated by the pure helper `tick(state, deltaMs)`, while `addLap` appends a lap at the current `progressMs(s)`. We can define timer-generated events as:

$$e_i = \text{Lap}(\ell_i) \quad \text{whenever} \quad \texttt{addLap} \text{ is invoked at elapsed time } t_i.$$

Session start and end times can be defined as:

$$t_{\text{start}} = 0, \qquad t_{\text{end}} = t_{\text{el}}^{\text{final}},$$

where $t_{\text{el}}^{\text{final}}$ is the final elapsed time when the clinician stops the timer. These timestamps anchor the entire timeline.

Flow events. Each flow run in the flows subsystem has a natural start and end hook. For a given flow with identifier $f \in \mathcal{F}$ (for example, `"agitation"`, `"safety"`, `"capacity"`), we define:

- At the moment the flow is launched (for example, when the clinician clicks "Start agitation flow"), record a $\text{FlowStart}(f)$ event at $t = t_{\text{el}}$, the current timer elapsed time.
- When the flow is completed and an outcome object $o \in \mathcal{O}_f$ is built (for example, by `agitationOutcome.ts`), record a $\text{FlowDone}(f, o)$ event at the then-current elapsed time.

The event payload o typically includes risk grades, interventions, and follow-up recommendations; incorporating it into e_i allows the timeline model to capture not just "when" a flow ran, but also "what" it concluded.

Other events. Intervention and documentation events can be attached to specific UI actions (for example, recording PRN medication from a flow step, or issuing an export from the Tools module). Each action occurs at a timer-defined elapsed time and yields an event of the appropriate kind.

13.2.4 Algorithm for constructing the timeline

Given access to the timer engine, flows host, and relevant UI actions, we can outline a generic procedure for constructing $\mathcal{T}$ at the end of a session:

1. Initialise an empty list of events.
2. While the session is in progress:
   (a) On each invocation of `addLap`, append $(t, \text{Lap}(\ell))$ to the event list, where t is the current elapsed time and ℓ is the lap label (if any).
   (b) On flow launch, append $(t, \text{FlowStart}(f))$.
   (c) On flow completion, append $(t, \text{FlowDone}(f, o))$.
   (d) On intervention or documentation actions, append the corresponding (t, e_i) pair.
3. When the session ends (timer stopped), set $T_{\text{end}} = t_{\text{el}}^{\text{final}}$ and sort the event list by time to obtain the sequence $\{(t_i, e_i)\}_{i=1}^{N}$.
4. Derive intervals I_i and labels L_i using the labelling function λ defined previously.

Because the timer engine is pure and deterministic, and because flows outcome builders are pure functions of their input states, this procedure can be re-run post hoc on persisted state (for example, in a supervision or quality-improvement context) to reconstruct the session timeline for further analysis or for improved narrative generation.

13.2.5 Visual schema of the timeline pipeline

Figure 40 presents a vertically organised, coloured schema of how timer and flows information is transformed into a session timeline and subsequently into narrative and analytic artefacts.

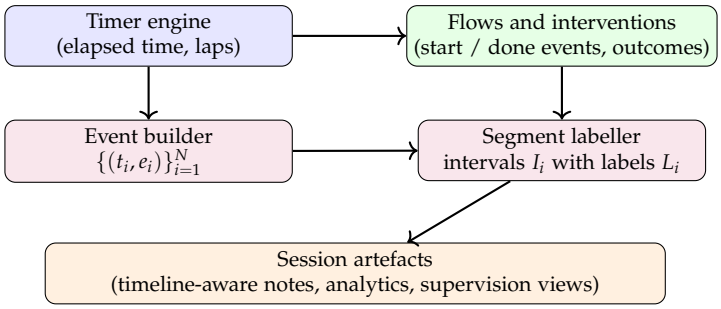


Figure 40: *Session timeline pipeline in a compact two-column layout. The timer engine and flows subsystem generate time-stamped events, which are transformed into labelled segments and then into session-level artefacts such as timeline-aware notes and analytics.*

By embedding this timeline model into the AI prompt-building and export logic, SynapseCore can produce narratives that are explicitly temporal (for example, structured around the phases of a crisis intervention) and can support research into how clinicians allocate time across different components of a psychiatric consultation.

13.3 Measurement-Based Care Formalisation

Section 6 described the measurement-based care (MBC) engine at the implementation level, focusing on the `ScoreResult` schema and the deterministic calculators in `src/features/psychiatry/mbc/mbc/calculators.ts`. In the present section we place this subsystem within a more general mathematical framework. We treat each instrument as a typed function on discrete response vectors, formalise the construction of composite indices and risk grades, and describe how these quantities are embedded into the session timeline and downstream learning pipelines.

At a high level, the formalisation proceeds in three layers:

1. *Instrument layer:* each scale (PHQ-9, GAD-7, PCL-5, Y–BOCS, AUDIT-C) is a deterministic function from a discrete response space to a structured score object, mirroring the `ScoreResult` type.
2. *Composite layer:* per-patient, per-time-point score objects are aggregated into a small set of composite indices (e.g. overall symptom burden) and mapped into discrete risk grades.
3. *Feature layer:* score objects and risk grades are embedded into a fixed-dimensional feature vector suitable for statistical analysis and machine learning (Section 14).

Figure 41 summarises this pipeline as a sequence of pure functions; Table 62 lists the core instruments and their domains.

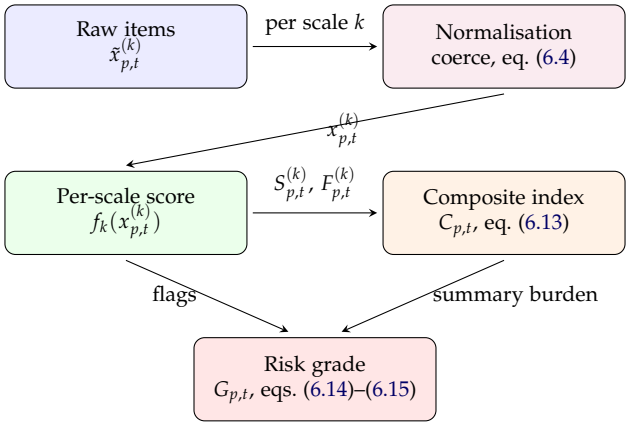


Figure 41: *Measurement-based care as a functional pipeline. Raw item arrays $\tilde{x}_{p,t}^{(k)}$ are normalised into bounded vectors $x_{p,t}^{(k)}$, scored by per-scale functions f_k, aggregated into a composite index $C_{p,t}$, and combined with deterministic flags to yield a discrete risk grade $G_{p,t}$.*

Table 62: *Core instruments in the SynapseCore measurement-based care engine. Domains and flags follow published manuals; see Section 6 for implementation details. Column widths are bounded to avoid overflow.*

Instrument	**Items** n_m	**Range** $0..k_m$	**Example flags / notes**
PHQ-9 (depression)	$n_{\text{PHQ9}} = 9$	$k_{\text{PHQ9}} = 3$	Item 9 suicidal ideation flag; five-band severity mapping (None, Mild, Moderate, Moderately severe, Severe).
GAD-7 (anxiety)	$n_{\text{GAD7}} = 7$	$k_{\text{GAD7}} = 3$	Simple sum score with four severity bands; no additional structural flags in the core implementation.
PCL-5 (PTSD)	$n_{\text{PCL5}} = 20$	$k_{\text{PCL5}} = 4$	Threshold at ≥ 33 for positive screen plus cluster predicates (B, C, D, E), exposed as a structured `breakdown`.
Y–BOCS (OCD)	$n_{\text{YBOCS}} = 10$	$k_{\text{YBOCS}} = 4$	Total score bands from Subclinical to Extreme; potential future extension to obsession/compulsion subscales.
AUDIT-C (alcohol use)	$n_{\text{AUDITC}} = 3$	$k_{\text{AUDITC}} = 4$	Sex-specific threshold for positive screen and a heavy episodic drinking flag based on the third item.

13.3.1 Scales as deterministic functions

We first formalise each instrument as a pure function on discrete response vectors, mirroring the implementation-level design based on `ScoreResult`. Let $\mathcal{M}$ denote the finite set of supported measures,

$$\mathcal{M} = \{\text{PHQ9}, \text{GAD7}, \text{PCL5}, \text{YBOCS}, \text{AUDITC}\},$$

and let $\mathcal{B}_m$ be the finite set of severity band labels associated with measure $m \in \mathcal{M}$.

For each instrument m, define the discrete response space

$$X_m = \{0, \ldots, k_m\}^{n_m},$$

where n_m is the number of items and k_m the highest admissible Likert category. Equation (6.3) in Section 6 introduced the generic item vector $x = (x_1, \ldots, x_n) \in \{0, \ldots, k\}^n$; here we simply make explicit the dependence on the instrument via indices n_m and k_m.

In the concrete implementation, `coerce` enforces bounds and length consistency on raw arrays $\tilde{x}$ coming from the UI or external systems, see equation (6.4). For each instrument m,

we define the normalised item vector at patient p and time t as

$$x_{p,t}^{(m)} = \mathrm{coerce}\big(\tilde{x}_{p,t}^{(m)}; n_m, 0, k_m\big) \in X_m.$$

Instrument-specific scoring maps. Given $x_{p,t}^{(m)} \in X_m$, the first component of the scoring logic is the integer-valued sum

$$S_m\big(x_{p,t}^{(m)}\big) = \sum_{i=1}^{n_m} x_{p,t,i}^{(m)}, \tag{13.1}$$

which is the instrument-specific version of equation (6.5). The total S_m lies in the interval $[0, n_m k_m] \cap \mathbb{N}$.

Severity banding is captured by an instrument-specific map

$$B_m : \mathbb{N} \to \mathcal{B}_m \cup \{\text{Unknown}\}, \tag{13.2}$$

which specialises the generic banding function B in equation (6.7). The band definition is represented in code by a `ScoreBand[]` array; the chosen severity label for a given score S is $B_m(S)$.

In addition to S_m and B_m, some instruments define structured flag maps and optional breakdowns. For each m we write

$$F_m : X_m \to 2^{\mathcal{F}_m}, \tag{13.3}$$

$$\beta_m : X_m \to \mathcal{B}_m^{\mathrm{aux}} \cup \{\varnothing\}, \tag{13.4}$$

where $\mathcal{F}_m$ is the finite set of string-valued flags for instrument m (e.g. suicidal ideation, heavy episodic alcohol use, PTSD cluster criteria) and $\mathcal{B}_m^{\mathrm{aux}}$ denotes an auxiliary space for structured breakdowns (such as the Boolean cluster indicators B, C, D, E in PCL-5). Both maps are implemented as pure functions over the normalised item vectors.

Putting these components together, each instrument can be viewed as a single deterministic scoring function

$$f_m : X_m \to \mathrm{ScoreResult}_m := \big(S_m,\, \sigma_m,\, \mathcal{B}_m,\, F_m,\, \beta_m\big), \tag{13.5}$$

where $\sigma_m(S) = B_m(S)$ selects the active severity band. The TypeScript signature

```
export function phq9Score(items: number[]): ScoreResult { ... }
```

is precisely the implementation of f_{PHQ9} with $X_m = \{0,\dots,3\}^9$ and codomain given by the concrete `ScoreResult` type.

Example: PHQ-9 as a function. For PHQ-9 we make the signature explicit. The domain is $X_{\mathrm{PHQ9}} = \{0,\dots,3\}^9$, and the codomain can be projected to the pair (S, σ) when we are only interested in the total and band label. Writing $\mathcal{B}_{\mathrm{PHQ9}}$ for the five PHQ-9 bands, we obtain

$$f_{\mathrm{PHQ9}} : \{0,\dots,3\}^9 \to \mathbb{N} \times \mathcal{B}_{\mathrm{PHQ9}}, \qquad f_{\mathrm{PHQ9}}(x) = \big(S_{\mathrm{PHQ9}}(x), B_{\mathrm{PHQ9}}(S_{\mathrm{PHQ9}}(x))\big), \tag{13.6}$$

together with an additional flag map that checks item 9:

$$F_{\mathrm{PHQ9}}(x) = \begin{cases} \{\text{“Item 9 > 0 — discuss safety today”}\}, & \text{if } x_9 > 0, \\ \varnothing, & \text{otherwise.} \end{cases}$$

This illustrates a general principle: all clinical predicates are encoded as simple, inspectable functions of the normalised item vector; no probabilistic state is introduced at the scoring stage.

13.3.2 Composite indices and risk grades

While (13.5) formalises each instrument in isolation, clinical decision-making often requires a cross-scale view of symptom burden and acute risk. SynapseCore therefore introduces composite indices built from per-scale scores and uses them as inputs to a discrete risk grading function.

Cross-scale composite indices. For a given patient p at time t, let $S_{p,t}^{(k)}$ denote the total score for scale $k \in \mathcal{K}$ obtained from the corresponding function f_k as in (13.1). Let M_k be the maximum possible score for scale k (e.g. $M_{\text{PHQ9}} = 27$), so that $S_{p,t}^{(k)} \in [0, M_k] \cap \mathbb{N}$. As detailed in equation (6.12), normalised scores are defined as

$$\tilde{S}_{p,t}^{(k)} = \frac{S_{p,t}^{(k)}}{M_k} \in [0,1].$$

Choosing non-negative weights $w_k \geq 0$ with $\sum_{k \in \mathcal{K}} w_k = 1$, the composite index $C_{p,t} \in [0,1]$ is given by equation (6.13),

$$C_{p,t} = \sum_{k \in \mathcal{K}} w_k \, \tilde{S}_{p,t}^{(k)}.$$

From a functional perspective, we can view this construction as a map

$$\Phi : \prod_{k \in \mathcal{K}} \text{ScoreResult}_k \to [0,1], \qquad \Phi\big((\text{SR}_{p,t}^{(k)})_{k \in \mathcal{K}}\big) = C_{p,t}, \tag{13.7}$$

where each $\text{SR}_{p,t}^{(k)}$ is the full score object returned by f_k and Φ implements the normalisation and weighting described above. In future variants, Φ can be enriched to incorporate non-linear transformations or shrinkage towards clinically meaningful anchors, while remaining a pure function on the vector of per-scale scores.

Risk grade mapping with flag adjustments. The composite index $C_{p,t}$ captures overall burden but does not encode structural risk cues such as suicidal ideation or severe PTSD clusters. Equation (6.14) therefore introduces a risk grade map

$$R : \mathbb{R}^d \to \{1,2,3,4,5\}, \qquad R\big(\mathbf{S}_{p,t}\big) = G_{p,t},$$

with $\mathbf{S}_{p,t}$ the vector of per-scale totals. In the simplest implementation, R factors through the composite index $C_{p,t}$ via the thresholds in equation (6.15).

To incorporate flags in a transparent way, we define the set of active flags at (p,t) as the union

$$F_{p,t} = \bigcup_{k \in \mathcal{K}} F_k(x_{p,t}^{(k)}) \subseteq \mathcal{F},$$

where $\mathcal{F} = \bigcup_{k \in \mathcal{K}} \mathcal{F}_k$ is the global flag alphabet. We then regard the risk grading map as a function of both the composite index and the flag set:

$$\Psi : [0,1] \times 2^{\mathcal{F}} \to \{1,2,3,4,5\}, \qquad \Psi\big(C_{p,t}, F_{p,t}\big) = G_{p,t}. \tag{13.8}$$

Conceptually, Ψ applies the threshold rule from equation (6.15) to $C_{p,t}$ and then applies deterministic adjustments based on critical flags. For example, a PHQ-9 item 9 flag may guarantee a minimum grade of $G_{p,t} \geq 4$, while severe alcohol flags from AUDIT-C may bump grades within the moderate-to-high range. All such adjustments are specified as simple, auditable predicates on $F_{p,t}$.

The `RiskLevel` type and helper functions in `src/centerpanel/rail/riskGrades.ts` implement the codomain of Ψ, mapping grades 1–5 to UI labels (e.g. "G1 • Low", "G4 • High") and to colour classes that can be reused in the measurement-based care views and longitudinal dashboards.

13.3.3 Embedding into the session timeline and feature space

The final layer of the formalisation concerns how measurement-based care outputs are embedded into the session timeline and exposed as features to downstream analytic components. This links the functional view of MBC to the session-level machine-learning agenda sketched in Section 14.

Longitudinal MBC snapshots. For each patient p and discrete time index t (e.g. encounter index or days since baseline), we define the MBC snapshot as the collection

$$\mathsf{MBC}_{p,t} = \big((\mathrm{SR}^{(k)}_{p,t})_{k\in\mathcal{K}},\ C_{p,t},\ G_{p,t}\big),$$

where $(\mathrm{SR}^{(k)}_{p,t})_{k\in\mathcal{K}}$ are the per-scale score objects from (13.5), $C_{p,t}$ is the composite index from (13.7), and $G_{p,t}$ is the risk grade from (13.8). At the data-structure level, this snapshot corresponds to a view over the `assessments[]` and `encounters[]` arrays, together with cached `ScoreResult` objects.

The longitudinal trajectory for patient p is then the sequence

$$\big(\mathsf{MBC}_{p,t}\big)_{t=0}^{T_p},$$

which can be used for descriptive analytics (e.g. plotting per-scale trajectories), for risk trajectory visualisation (grades over time), or as input to survival-like models of relapse and response.

Feature maps for learning components. To integrate measurement-based care into learning components (e.g. classifiers for acute risk or predictors of treatment response), we require a finite-dimensional feature vector $\mathbf{z}_{p,t}$ derived from $\mathsf{MBC}_{p,t}$. This is achieved via per-scale feature maps.

For each scale k, let $\mathcal{B}_k = \{\ell^{(k)}_1, \dots, \ell^{(k)}_{K_k}\}$ be its ordered set of severity labels and let $\mathcal{F}_k$ be its flag alphabet. We define a basic feature map

$$\phi_k : \mathrm{ScoreResult}_k \to \mathbb{R}^{q_k}, \tag{13.9}$$

constructed from:

- the normalised score $\tilde{S}^{(k)}_{p,t} \in [0,1]$,
- one-hot encodings of the active severity band $\sigma^{(k)}_{p,t} \in \mathcal{B}_k$,
- binary indicators for a small, curated subset of flags in $\mathcal{F}_k$ (e.g. suicidal ideation, heavy episodic use, PTSD cluster criteria).

A simple example of such a map is

$$\phi_k\big(\mathrm{SR}^{(k)}_{p,t}\big) = \Big(\tilde{S}^{(k)}_{p,t}, \mathbf{1}\{\sigma^{(k)}_{p,t} = \ell^{(k)}_1\}, \dots, \mathbf{1}\{\sigma^{(k)}_{p,t} = \ell^{(k)}_{K_k}\}, \mathbf{1}\{f \in F^{(k)}_{p,t}\}_{f\in\mathcal{F}^\star_k}\Big),$$

where $\mathcal{F}^\star_k \subseteq \mathcal{F}_k$ is a selected subset of flags and $\mathbf{1}\{\cdot\}$ denotes the indicator function.

The global MBC feature vector at (p, t) is then obtained by concatenating per-scale features with the composite index and risk grade:

$$\mathbf{z}_{p,t} = \left(\phi_k(\mathrm{SR}_{p,t}^{(k)})\right)_{k\in\mathcal{K}} \oplus C_{p,t} \oplus G_{p,t} \in \mathbb{R}^q, \tag{13.10}$$

where $q = \sum_{k\in\mathcal{K}} q_k + 2$ and $\oplus$ denotes vector concatenation. In the codebase, this corresponds to a pure helper that:

1. retrieves or computes `ScoreResult` objects for each scale via the calculators in `src/features/psychiatry/mbc/calculators.ts`,
2. applies a per-scale feature map (the analogue of ϕ_k) to each result,
3. computes the composite index $C_{p,t}$ using a `computeCompositeIndex` helper consistent with equation (6.13),
4. maps $C_{p,t}$ and $F_{p,t}$ to a risk grade $G_{p,t}$ via a pure function implementing Ψ in (13.8),
5. concatenates all components into a typed feature vector ready for consumption by TensorFlow.js models or other learning backends.

By keeping all maps in equations (13.5), (13.7), (13.8), and (13.10) pure and side-effect free, SynapseCore ensures that measurement-based care plays a transparent, reproducible role in any downstream analytics. The formalisation here provides the bridge between the implementation-level MBC engine of Section 6 and the session-level learning components described in Section 14.

13.4 AI Orchestration Semantics

At run time the AI layer of SynapseCore mediates between three kinds of objects: (i) human inputs such as free–text questions or clinical commands, (ii) a typed, multi–module context derived from the psychiatry feature, flows, timer, and IDE subsystems, and (iii) probabilistic language models exposed by different providers. This subsection makes the orchestration semantics explicit, so that the behaviour of the system can be understood as a well defined family of mappings between these objects rather than as opaque "prompt magic".

Spaces and notation. Let Σ denote the token alphabet of a given provider family and Σ^* the set of all finite token sequences. For any countable set X we write $\mathrm{Dist}(X)$ for the set of discrete probability measures over X.[3] User utterances live in a space $\mathcal{U}$ (plain Unicode strings), while the internal clinical and technical state at a given time t is captured by a context space $\mathcal{C}$. The active AI route—provider, model, and sampling configuration at time t—is an element of a finite set $\mathcal{R}$, already introduced in the observability section through the component $\mathcal{R}(t)$ of the system tuple $\mathcal{S}(t)$.

To reason about the orchestration layer we treat the following objects as first–class:

- A *route* $r \in \mathcal{R}$, which encodes a provider $p(r)$, model identifier $m(r)$ and sampling configuration $s(r)$.[4]
- A *context bundle* $C \in \mathcal{C}$, assembled from the psychiatry store, active flow state, timer laps, and IDE selection state.
- A *conversation history* $H = \big((r_i, u_i, y_i)\big)_{i=1}^{n}$, where each $u_i \in \mathcal{U}$ is a user turn and $y_i \in \Sigma^*$ the corresponding assistant answer produced under route r_i. On the implementation

[3] In the architectural discussion of Section 3 we wrote $\mathcal{D}(X)$ for this set; here we make the measure–theoretic interpretation explicit by using $\mathrm{Dist}(X)$.

[4] At code level this aligns with the route stored in `useAiConfigStore`, which combines a `ProviderId`, model name, and `Sampling` record.

side H corresponds to a list of `ChatMsg` objects persisted by `chatPersistence.ts` and exposed to the UI via `useSynapseChat`.

Prompt construction as a function. Given a raw user utterance $u \in \mathcal{U}$, a context bundle $C \in \mathcal{C}$, and a route $r \in \mathcal{R}$, the first step is to construct a concrete prompt string $x \in \Sigma^*$.[5] Abstractly we define a prompt constructor

$$P : \mathcal{U} \times \mathcal{C} \times \mathcal{R} \longrightarrow \Sigma^*, \qquad (u, C, r) \longmapsto x, \tag{13.11}$$

which factors into three logical stages:

1. *Context projection.* A function $\Pi : \mathcal{C} \times \mathcal{R} \to \mathcal{C}'$ extracts exactly those components of the full context that are relevant for the route r. For example, a summarisation route for the psychiatry modal will include MBC scores, risk flags, and flow summaries, whereas an IDE refactoring route will include file content and selections but no clinical data.
2. *Template application.* A route–specific template T_r combines the projected context $\Pi(C, r)$ and the new utterance u into a draft prompt body $b_r(u, \Pi(C, r))$ which still lives in a higher–level structured space (for example, a TypeScript object with named fields such as `instruction`, `clinicalContext`, and `workspaceContext`).
3. *Serialisation.* A serialiser S maps the structured body to an actual token sequence, $S : \text{Body}_r \to \Sigma^*$, yielding $x = S\big(b_r(u, \Pi(C, r))\big)$.

In code these three stages correspond respectively to: (i) gathering typed context from stores such as the psychiatry store, flows store, timer hooks, and IDE context builders; (ii) route–specific helper functions that assemble a canonical "prompt object"; and (iii) the message construction logic passed into the provider adapters via `useAiStreaming` or, for simple OpenAI routes, `useSimpleOpenAIStream`.

Provider kernels as probabilistic transducers. For a fixed provider p and model m we view the underlying language model as a probabilistic transducer

$$\mathcal{M}_{p,m} : \Sigma^* \times \mathcal{C}' \longrightarrow \mathsf{Dist}(\Sigma^*), \qquad (x, C') \longmapsto \mu_{p,m}^{x,C'}, \tag{13.12}$$

where x is the serialised prompt and C' is the projected context actually given to the adapter (for example, JSON mode flags or tool definitions). A single sample Y from this kernel is a random response sequence

$$Y \sim \mathcal{M}_{p,m}(x, C') \in \Sigma^*.$$

The sampling mapper described earlier formalises how a generic sampling configuration $s(r)$ and route meta–data $c(r)$ are turned into provider–specific parameters and metadata. Recall the transformation

$$\Phi_p : \mathcal{S} \times \mathcal{P} \longrightarrow \Theta_p \times \mathcal{M},$$

where $\mathcal{S}$ is the abstract sampling space, $\mathcal{P}$ a space of provider meta–data, and Θ_p the concrete parameter space of provider p. Given a route r with provider $p(r)$ and sampling configuration $s(r)$, the adapter obtains

$$(\theta_{p(r)}, m_{p(r)}) = \Phi_{p(r)}(s(r), c(r)),$$

and calls the appropriate streaming or completion method with parameters $\theta_{p(r)}$. Equation (13.12) then describes the induced distribution on completions for that route.

[5] In practice this string may mix a system prompt, instruction header, and conversation transcript; for expository clarity we treat it as a single sequence x.

Streaming semantics. The TypeScript adapters implement streaming via a stream of *delta events* and a final *done event*. Mathematically we can recover this behaviour by considering a *prefix process* derived from Y. Let $Y = (y_1, \dots, y_L)$ be the random token sequence produced by the kernel at (13.12). A streaming run is realised as a sequence of random variables

$$\Delta_1, \Delta_2, \dots, \Delta_K,$$

each taking values in Σ^*, such that their concatenation recovers Y:

$$Y = \Delta_1 \,\|\, \Delta_2 \,\|\, \cdots \,\|\, \Delta_K, \tag{13.13}$$

where $\|$ denotes string concatenation. After receiving the k–th delta the UI maintains a running prefix

$$\hat{Y}_k = \Delta_1 \,\|\, \cdots \,\|\, \Delta_k, \qquad 1 \le k \le K,$$

and, by construction, $\hat{Y}_K = Y$ almost surely.

The hook `useAiStreaming` implements an equivalent operational semantics:

1. When a new job is submitted, `useSynapseChat` appends an assistant placeholder message m^{asst} to the history H and marks it as *streaming*.
2. Each delta event from the adapter triggers a call to `mergeAssistantDelta`, which updates the content field of m^{asst} by appending the new chunk. Operationally this is just the update $\hat{Y}_{k+1} \leftarrow \hat{Y}_k \,\|\, \Delta_{k+1}$.
3. When the done event is received, the hook calls `finalizeAssistant` (or the route–aware variant `finalizeLatestStreamingAssistant`), which marks the message as non–streaming, attaches token and cost estimates, and records any finish reason exposed by the adapter.

An error event instead triggers `setErrorOnAssistant`, which preserves the partial prefix $\hat{Y}_k$ but annotates the message with an error field so that the UI can display both the partial content and a human–readable explanation of what went wrong.

Conversation–level semantics. It is often more convenient to reason about the AI behaviour at the level of entire turns rather than individual streams. A single interaction step takes as input: (i) the current history H, (ii) a new utterance u, and (iii) a route r chosen by the user or derived from defaults. We write this step as a stochastic transition

$$F_{\text{ai}} : \mathcal{H} \times \mathcal{U} \times \mathcal{C} \times \mathcal{R} \longrightarrow \mathsf{Dist}(\mathcal{H}), \qquad (H, u, C, r) \longmapsto \nu_{H,u,C,r}, \tag{13.14}$$

where $\mathcal{H}$ is the space of finite histories. Sampling a new history $H' \sim \nu_{H,u,C,r}$ corresponds to sampling a completion $Y \sim \mathcal{M}_{p(r),m(r)}(P(u, C, r), C')$ and appending it as a new assistant turn with appropriate route meta–data:

$$H' = H \,\|\, \big(r, u, Y\big).$$

From the implementation perspective, the transition $H \mapsto H'$ is realised by the interplay of three components:

1. **Chat state machine.** The finite–state machine in `useChatFSM` controls when a new job may be launched (for example, it prevents overlapping streams on the same route), and tracks whether the panel is idle, sending, or streaming.
2. **Streaming hook and adapters.** `useAiStreaming` coordinates the asynchronous execution of the probabilistic kernel through provider adapters, converts stream events into deterministic updates on the React state, and emits telemetry events for observability.

Table 63: *Semantic objects in the AI orchestration layer and their main code–level representations.*

Symbol / object	Semantic role	Representative types / hooks	Primary module
$r \in \mathcal{R}$	Active AI route (provider, model, sampling)	`ProviderId`, `Sampling`	`useAiConfigStore`
$C \in \mathcal{C}$	Typed clinical / IDE context bundle	Aggregated store state, IDE context builders	Psychiatry store, flows, timer, IDE
P from (13.11)	Prompt constructor mapping (u, C, r) to x	Route–specific prompt helpers	AI panel, psychiatry AI utilities
$\mathcal{M}_{p,m}$ from (13.12)	Provider kernel (probabilistic transducer)	`Adapter`, `StreamEvent`	AI adapters and HTTP layer
F_{ai} from (13.14)	Conversation–level transition on histories H	`useChatFSM`, `useSynapseChat`	AI panel state machine
$\widehat{\mathcal{M}}_r$	Effective kernel with fallback routing	Job queue and retry logic	`useAiStreaming`

3. **Persistence layer.** `chatPersistence.ts` maintains the durable history in localStorage, so that the abstract history H above survives reloads and is available to both the AI panel and the psychiatry feature.

The combination of these pieces guarantees that, for any concrete realisation of the random variables in equations (13.12)–(13.14), the UI and persistence layer represent a single coherent history that can be handed back into the prompt constructor P on the next turn.

Multi–provider orchestration and fallbacks. The orchestration layer also implements a simple yet clinically useful policy for multi–provider routing. Given a route r with primary provider $p(r)$, the hook computes an ordered list of candidate providers $(p_1, \dots, p_J)$ where $p_1 = p(r)$ and the remaining providers encode a fallback ordering (for example, trying Gemini or Anthropic if OpenAI is temporarily unavailable). Operationally this is implemented by the job queue in `useAiStreaming`, which attempts to execute the job with p_1 and, on certain classes of transport or quota errors, resubmits the same logical job to p_2, and so on.

From the semantic point of view this can be seen as lifting the kernel $\mathcal{M}_{p,m}$ to a mixture over providers. Let $\mathcal{E}$ denote the event that a request fails with a retry–eligible error (for example, HTTP 5xx or connection timeout). For a fixed logical route r and candidate list $(p_1, \dots, p_J)$ we can write the effective kernel as

$$\widehat{\mathcal{M}}_r = \mathcal{M}_{p_1,m(r)}\, \mathbf{1}_{\neg\mathcal{E}} + \sum_{j=2}^{J} \mathcal{M}_{p_j,m(r)}\, \mathbf{1}_{\mathcal{E}_{j-1} \wedge \neg\mathcal{E}_j},$$

where $\mathcal{E}_j$ is the event that all attempts up to p_j failed. Although the current implementation only uses this mechanism for resilience rather than for ensembling, making the mixture explicit is helpful when reasoning about audit logs and reproducibility: repeated calls with identical (H, u, C, r) may traverse different providers in the presence of transient infrastructure errors.

Taken together, equations (13.11)–(13.14), Table 63 and Figure 42 provide a compact but precise description of how the AI orchestration layer behaves. Crucially, the semantics are expressed in terms of typed functions and probability kernels that have direct counterparts in the TypeScript code; this makes the system amenable to formal reasoning about reliability, reproducibility, and clinical safety, and provides a clear bridge between the informally understood notion of "asking the AI a question" and the concrete sequence of state updates implemented in SynapseCore.

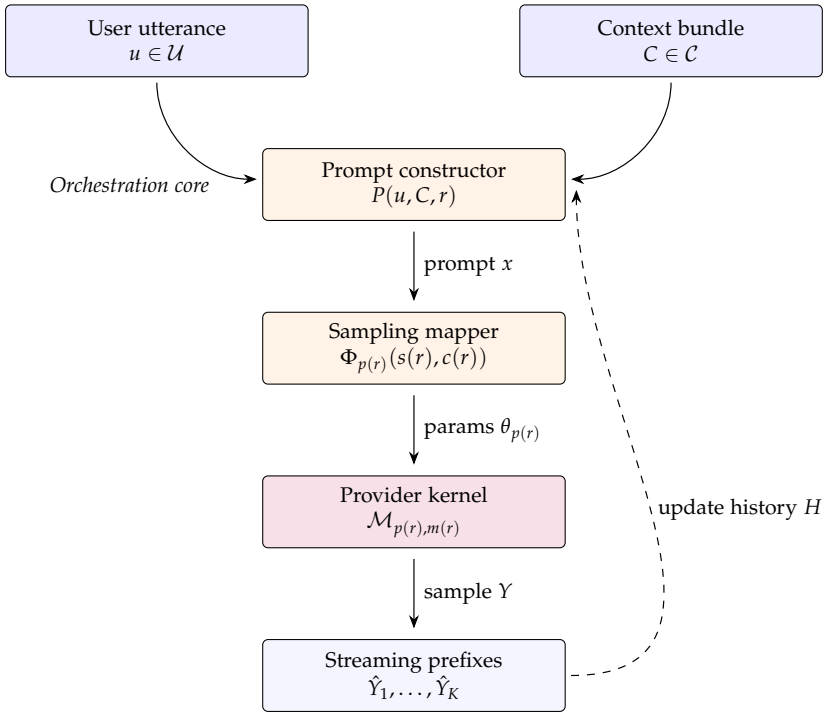


Figure 42: *Schematic view of the AI orchestration semantics: user utterance u, context C, and route r are mapped by P to a prompt x, which is passed through the sampling mapper $\Phi_{p(r)}$ and provider kernel $\mathcal{M}_{p(r),m(r)}$. Streaming produces prefixes $\hat{Y}_k$ that are fed back into the conversation history for the next turn.*

13.5 Flows as Finite State Machines

The flows subsystem described in Section 7.5 treats each clinical flow as a small directed acyclic graph with typed data domains and an accumulator map ρ_F (Equation (7.4)). For implementation and verification, it is equally useful to view the same object as a *finite state machine* (FSM) that processes a stream of user events and emits both UI updates and clinical documentation artefacts. This subsection makes that connection explicit and links the TypeScript implementation in the flow shells (e.g. agitation, safety, lorazepam challenge) to a standard FSM formalism.

13.5.1 From flow DAGs to input–output state machines

Let F be a fixed flow (for example, the agitation flow implemented in `AgitationFlowShell.tsx`) with DAG representation $(V_F, E_F, v_F^{(0)}, V_F^{\text{term}}, \mathcal{D}_F, \lambda_F, \rho_F)$ as in Equation (7.4). The corresponding finite state machine is defined as

$$\mathcal{M}_F = \big(Q_F,\, \Sigma_F,\, \Gamma_F,\, \delta_F,\, \omega_F,\, q_F^{(0)},\, F_F\big), \tag{13.15}$$

with the following components:

- Q_F is the finite set of control states and is taken to be identical to the node set V_F. Each $q \in Q_F$ denotes a single logical step of the flow.
- Σ_F is the *input alphabet*, consisting of abstract user actions such as next, back, $\mathsf{jump}(k)$, $\mathsf{edit}(v, f, x)$, and submit. These inputs correspond to button presses, step pill clicks, and form edits in the React shell components.
- Γ_F is the *output alphabet*, representing observable effects such as UI state changes (e.g. highlighting a new step) and side-effectful operations (e.g. appending a completed run to the encounter record or inserting structured text into the note).

Table 64: *Mapping between the DAG-oriented flow structure and its finite state machine realisation. The FSM view emphasises event handling and output behaviour while reusing the same underlying step set V_F.*

Aspect	Formal role	Implementation hook
Q_F / V_F (nodes)	Control states of the FSM	Entries in the constant `STEPS` array in each shell component (e.g. agitation, safety, lorazepam), plus types such as `StepPill`.
Σ_F (inputs)	Event alphabet driving transitions	Normalised user actions: `Next`, `Back`, step pill selection, form edits, and the final "Insert" / "Mark run completed" actions.
Γ_F (outputs)	Output alphabet (UI + side effects)	UI redraws (active step highlight), validation messages, writes into the encounter registry, and insertion of neutral narratives into the note.
δ_F (transition function)	Deterministic state evolution $Q_F \times \Sigma_F \to Q_F$	Index arithmetic and clamping in helper functions such as `goNext` and `goBack` in flow shells, constrained by the step ordering induced by `STEPS`.
ω_F (output function)	Mealy-style output map $Q_F \times \Sigma_F \to \Gamma_F$	Event handlers that both change the active step and trigger effects, for example the call to a builder in `Flows/builders/*.ts` when the user finalises a run.
$q_F^{(0)}, F_F$ (initial/terminal states)	Start and accepting states	First entry of `STEPS` (initial step) and one or more "Outcome" / "Summary" steps that unlock finalisation controls and produce completed runs for later review.

- $\delta_F : Q_F \times \Sigma_F \to Q_F$ is the deterministic transition function, governing how the active step changes in response to each user action.
- $\omega_F : Q_F \times \Sigma_F \to \Gamma_F$ is the output function (Mealy-style), encoding which UI and documentation updates are triggered by each transition.
- $q_F^{(0)} = v_F^{(0)}$ is the distinguished initial state corresponding to the first step in the flow.
- $F_F \subseteq Q_F$ is the set of *accepting* (terminal) states, which coincides with V_F^{term} from the DAG representation. Once the machine enters one of these states and a submit input is processed, a completed flow run is produced.

Under this view, the graph structure (V_F, E_F) encodes which transitions are even *admissible*, whereas the finite state machine $\mathcal{M}_F$ describes the operational semantics of traversing that graph in response to live input, including the side effects that yield concrete clinical documentation.

13.5.2 Event alphabet and transition function

The finite input alphabet Σ_F abstracts away from low-level DOM events and focuses on a small set of clinically meaningful actions:

$$\Sigma_F = \{\mathsf{next},\ \mathsf{back},\ \mathsf{jump}(k),\ \mathsf{edit}(v, f, x),\ \mathsf{submit}\}, \tag{13.16}$$

where $\mathsf{jump}(k)$ denotes a direct jump to step index k, $\mathsf{edit}(v, f, x)$ denotes an edit to field f at node v with value x, and submit denotes the act of trying to finalise the flow.

In the current implementation, the forward and backward transitions are almost entirely governed by simple index arithmetic over the `STEPS` array. If the steps are enumerated as

$(s_0, s_1, \dots, s_{n-1})$, the transition function for these actions can be written as

$$\delta_F(s_i, \mathsf{next}) = s_{\mathrm{clamp}(i+1,\,0,\,n-1)}, \tag{13.17}$$

$$\delta_F(s_i, \mathsf{back}) = s_{\mathrm{clamp}(i-1,\,0,\,n-1)}, \tag{13.18}$$

$$\delta_F(s_i, \mathsf{jump}(k)) = s_{\mathrm{clamp}(k,\,0,\,n-1)}, \tag{13.19}$$

where the clamping function corresponds exactly to the TypeScript helper routinely used in the shells:

$$\mathrm{clamp}(i, 0, n-1) = \min\{n-1,\, \max\{0,\, i\}\}. \tag{13.20}$$

This ensures that even when users repeatedly press `Next` or `Back` at the edges of the flow, the state machine remains within the valid index range and no illegal transitions are taken.

For edit events, the control state does not change:

$$\delta_F(s_i, \mathsf{edit}(v, f, x)) = s_i, \tag{13.21}$$

but the accompanying data update (formalised below) will modify the typed form state associated with step s_i. For submit, the transition is likewise typically self-looping:

$$\delta_F(s_i, \mathsf{submit}) = s_i, \tag{13.22}$$

with acceptance and output behaviour controlled via validation predicates and ω_F rather than by moving to a new state.

13.5.3 Extended configurations and data-carrying transitions

The DAG-based formalisation already introduced local data domains $\mathcal{D}_F = \{D_v : v \in V_F\}$ and flow runs as sequences (v_k, d_k) (Equation (7.5)). To connect this to the FSM view, it is convenient to lift the machine from pure control states Q_F to *configurations* of the form

$$c = (q, d),$$

where $q \in Q_F$ and $d \in D_q$ is the local data object associated with the active step.[6]

An extended transition relation over configurations is then defined by

$$(q, d) \xrightarrow{\alpha} (q', d') \quad \Longleftrightarrow \quad q' = \delta_F(q, \alpha) \text{ and } d' = u_F(q, \alpha, d), \tag{13.23}$$

where $u_F : Q_F \times \Sigma_F \times \bigcup_{v \in V_F} D_v \to \bigcup_{v \in V_F} D_v$ is an update function wrapping the form-state mutation logic. In the current codebase the updates are implemented in idiomatic React style:

- edits invoke `setForm` with an updater function that returns a new TypeScript object with the relevant field changed, and
- step changes call `setStep` with a clamped index, sometimes accompanied by minor adjustments to default values or helper-derived flags.

Equation (13.23) simply abstracts this into a single-step semantics for the flow: each user action α induces a precisely defined transition from one configuration to another.

Composing such steps yields precisely the notion of a *run* already used in Section 7.5. A finite sequence of inputs $\alpha_0, \dots, \alpha_T \in \Sigma_F$ generates a sequence of configurations

$$(q_0, d_0),\ (q_1, d_1),\ \dots,\ (q_T, d_T)$$

via repeated application of Equation (13.23), with $q_0 = q_F^{(0)}$. The projection onto the first component recovers the path in the DAG, and the second component recovers the local data at each step.

[6] In the implementation, the React state for a flow shell usually holds a single aggregate TypeScript object (e.g. `AgitationFormState`), but conceptually this can be decomposed into per-step slices D_v.

13.5.4 Termination, acceptance, and documentation output

An FSM run is *accepting* when the control state lies in the accepting set and all local validation checks are satisfied. For the flows subsystem, a run is considered clinically complete when two conditions hold:

1. The current control state corresponds to a terminal step: $q_T \in F_F = V_F^{\text{term}}$.
2. The local data recorded along the path satisfy per-step predicates $\text{valid}_v : D_v \to \{0,1\}$, as described in Section 7.5.

Equivalently, writing r for the run and reusing the $\text{complete}(r)$ notation,

$$\text{accept}(r) = 1 \quad \Longleftrightarrow \quad \text{complete}(r) = 1, \tag{13.24}$$

so that the acceptance condition of the FSM exactly matches the previously defined notion of a complete flow instance.

On acceptance, the output function ω_F is responsible for emitting a composite output symbol in Γ_F that encodes at least two components:

- a structured summary object $x_F \in X_F$, obtained via the accumulator $\rho_F(r)$, and
- a neutral natural-language narrative π_F suitable for insertion into the encounter note, obtained via the domain-specific builder in `Flows/builders/*.ts`.

In practice, the ω_F map is realised by the event handler bound to the finalisation control (e.g. "Insert in note") in the shell component: it reads the current form state, computes $\rho_F(r)$ and π_F, appends a new entry to the `completedRuns` list of the selected encounter, and inserts the textual narrative into the editor at the current selection.

13.5.5 Example: agitation flow as a finite state machine

The agitation flow shell illustrates how a clinically meaningful workflow is encoded as a small, typed FSM in idiomatic React/TypeScript. The shell declares a constant array of steps:

```
const STEPS = [
  { key: "baseline",    label: "Baseline Behavior" },
  { key: "deescalation", label: "De-escalation Attempts" },
  { key: "escalation",   label: "Escalation Rationale" },
  { key: "outcome",      label: "Monitoring / Outcome" },
] as const;
type StepIndex = 0 | 1 | 2 | 3;
```

together with a numeric step index and standard navigation helpers:

```
const clamp = (n: number, min: number, max: number) =>
  Math.max(min, Math.min(max, n));

const goNext = () =>
  setStep((s) => clamp(s + 1, 0, STEPS.length - 1) as StepIndex);

const goBack = () =>
  setStep((s) => clamp(s - 1, 0, STEPS.length - 1) as StepIndex);
```

Mathematically, the state set is $Q_F = \{q_{\text{baseline}}, q_{\text{deescalation}}, q_{\text{escalation}}, q_{\text{outcome}}\}$, with the obvious bijection between `StepIndex` values and control states. The transition function δ_F is induced

by the goNext and goBack helpers together with the step pill component StepPills, which implements jump(k) transitions.

The data domains D_v are slices of the AgitationFormState type, with fields such as objectiveBehavior, deescalationTechniques, and escalationRationale logically owned by different steps. When the clinician completes the "Outcome" step and chooses to insert the summary, the builder buildAgitationOutcome in Flows/builders/agitationOutcome.ts is invoked. It maps the final configuration into a neutral narrative and a structured record, which are then propagated through the registry and documentation layers exactly as described above.

13.5.6 Compositional view with other state machines

Finally, casting flows as finite state machines makes it straightforward to compose them with the other stateful subsystems introduced in Section 13. The session timer engine (Section 8) is itself a pure state machine over timer states, and the AI orchestration pipeline (Section 13.4) can also be viewed as a stochastic transducer from prompt strings to token streams.

At the level of formal modelling, a single consultation can therefore be described by the synchronous product of three interacting machines:

- the timer machine $\mathcal{T}$, which governs temporal evolution and lap events;
- the flow machine $\mathcal{M}_F$, which governs the progression of structured clinical documentation; and
- the AI orchestration machine $\mathcal{M}_{\text{AI}}$, which governs sequences of prompt–response exchanges.

The global system evolves over a composite state $(\sigma_{\mathcal{T}}, \sigma_{\mathcal{M}_F}, \sigma_{\mathcal{M}_{\text{AI}}})$, with user actions and timer ticks driving transitions in each component. This compositional view provides a precise mathematical backbone for future analysis of safety properties (e.g. "a suicide risk flow must reach an accepting state before discharge is recorded") and for the design of automated checks or simulations over synthetic clinical sessions.

14 Session ML and Learning Components

The session timer in SynapseCore is not only a time-keeping utility but also a sensor that produces structured traces of the clinical workday. Whenever a psychiatrist starts, pauses, segments, or completes a session, the timer engine (Section 8.1) emits a sequence of events that can be interpreted as a discrete-time approximation of the consultation process. The session ML layer builds on this by learning regularities in how sessions are scheduled, structured, and concluded, using only de-identified timing and segment metadata.

From a systems perspective, the learning components occupy a narrow but well-defined position between the deterministic timer engine and the user-facing coaching and analytics widgets:

- Downstream, they consume HistoricalSession objects saved into the browser and live SessionPattern snapshots exposed by useSessionML.
- Upstream, they expose SessionPrediction objects and strings returned by getProactiveSuggestion, which the UI renders in the timer modal, calendar heatmaps, and longitudinal dashboards.

The mathematical backbone is deliberately modest: a fixed feature map from session trajectories into $\mathbb{R}^d$ (this subsection), followed by a small TensorFlow.js model which learns

a conditional mapping from features to suggested next segment type and duration (detailed in later subsections). By keeping the feature map ϕ simple, bounded, and well motivated, we can make explicit claims about what information the model *does* and *does not* see, thereby supporting clinical interpretability and privacy.

14.1 Feature Engineering for Session Patterns

14.1.1 Session trajectories and event representation

We first formalise how a timer session is represented for the purposes of learning. Let $\mathcal{L}$ denote a finite set of segment labels, for example

$$\mathcal{L} = \{\text{Assessment, Therapy, Documentation, Break, Consultation, Other}\},$$

which corresponds directly to the codomain of `encodeSegmentName` and the categories used by `decodeSegmentType` in `centerpanel/timerHooks/useSessionML.ts`.

A single completed timer run is represented as a tuple

$$s = \big(t_0,\, T,\, (\ell_j, \Delta_j)_{j=1}^{n},\, L,\, P\big),$$

where

- $t_0 \in \mathbb{R}$ is the wall-clock start time (Unix timestamp in milliseconds);
- $T \geq 0$ is the total duration of the session (in seconds);
- $(\ell_j)_{j=1}^{n} \subseteq \mathcal{L}$ are segment labels;
- $(\Delta_j)_{j=1}^{n} \subseteq [0, \infty)$ are segment durations such that $\sum_{j=1}^{n} \Delta_j = T$;
- $L \in \mathbb{N}$ is the lap count and $P \in \mathbb{N}$ the pause count.

We can equivalently think of s as a piecewise-constant labelled process

$$\Lambda_s : [0, T] \longrightarrow \mathcal{L}, \qquad \Lambda_s(t) = \ell_j \;\text{ for }\; t \in \big[\tau_{j-1}, \tau_j\big),$$

with breakpoints $\tau_0 = 0 < \tau_1 < \cdots < \tau_n = T$ and $\Delta_j = \tau_j - \tau_{j-1}$. The timers and UI never manipulate Λ_s directly; instead they store the finite tuple (ℓ_j, Δ_j) together with L and P inside the TypeScript interface `HistoricalSession`:

```
interface HistoricalSession {
  timestamp: number;
  segments: Array<{ name: string; duration: number }>;
  totalDuration: number;
  lapCount: number;
  pauseCount: number;
}
```

Live sessions are represented as partial trajectories

$$s^{\text{live}} = \big(t_0,\, T_{1:k},\, (\ell_j, \Delta_j)_{j=1}^{k},\, L_{1:k},\, P_{1:k}\big),$$

where only the first k segments and cumulative statistics up to the current timer state are available. These correspond to the `SessionPattern` interface:

Table 65: *Vertically organised overview of the session feature map ϕ_{sess} from (14.6), grouped into clinically interpretable categories. Each block corresponds to a subset of the eight-dimensional feature vector used by the TF.js model.*

Feature group	Definition and interpretation
Temporal context	$h(s) = \text{hour}(s)/24$, $d(s) = \text{dow}(s)/7$. Together, these encode when the session starts within the weekly calendar and allow the model to differentiate, for example, early-morning intake clinics from late-evening on-call work.
Structural workload	$\bar{\tau}(s) = T(s)/(\max(1, n(s))\,3600)$, $N(s) = n(s)/10$. These summarise how work is parcelled into segments: many short segments with small $\bar{\tau}(s)$ and larger $N(s)$ indicate fragmented sessions, whereas few long segments indicate more consolidated work.
Interaction dynamics	$L^*(s) = L(s)/10$, $P^*(s) = P(s)/5$. Scaled counts of laps and pauses, capturing the degree of fragmentation and interruption. High values may correspond to complex consultations, frequent phone calls, or teaching sessions with more annotations.
Content phase	$I_{\text{ther}}(s)$, $I_{\text{assess}}(s)$, defined as substring indicators over segment names. These binary flags express whether therapy- or assessment-like segments are present, providing a coarse phase-of-care signal without exposing raw clinical content.

```
export interface SessionPattern {
  timeOfDay: number;
  dayOfWeek: number;
  previousSegments: string[];
  segmentDurations: number[];
  totalDuration: number;
  lapCount: number;
  pauseCount: number;
}
```

We write $\mathcal{S}_{\text{hist}}$ and $\mathcal{S}_{\text{live}}$ for the sets of all such completed and live trajectories, and

$$\mathcal{S} = \mathcal{S}_{\text{hist}} \cup \mathcal{S}_{\text{live}}$$

for their union. The feature map $\phi : \mathcal{S} \to \mathbb{R}^d$ is then defined on this common session space.

14.1.2 Decomposition of the feature map

Rather than constructing ϕ as a monolithic vector, we decompose it into four groups of interpretable components,

$$\phi_{\text{sess}} = \big(\phi_{\text{time}}, \phi_{\text{structure}}, \phi_{\text{interaction}}, \phi_{\text{content}}\big), \qquad \phi_{\text{sess}} : \mathcal{S} \to \mathbb{R}^8, \tag{14.1}$$

with each sub-map contributing a small number of bounded coordinates. This mirrors the implementation strategy in `extractFeatures(session, currentTime)` and the inline feature construction in `predictNextSegment`, which both return an 8-dimensional array.

Formally, for any $s \in \mathcal{S}$ we define:

(1) Temporal context. Let $\text{hour}(s) \in \{0, \dots, 23\}$ and $\text{dow}(s) \in \{0, \dots, 6\}$ denote, respectively, the local hour-of-day and day-of-week associated with the session start time t_0. The

temporal sub-map is

$$\phi_{\text{time}}(s) = \Big(h(s),\, d(s)\Big) = \left(\frac{\text{hour}(s)}{24},\, \frac{\text{dow}(s)}{7}\right) \in [0,1]^2. \tag{14.2}$$

Encoding hour and day as fractions in $[0,1]$ keeps the scale uniform and aligns exactly with the JavaScript expressions `hour / 24` and `dayOfWeek / 7`. Clinically, these coordinates allow the model to distinguish between, for instance, early-morning intake clinics, mid-afternoon therapy blocks, and late-evening on-call work.

(2) Structural workload. Let $n(s)$ be the number of segments in s and $T(s)$ the total duration in seconds. We define the mean segment duration in hours as

$$\bar{\tau}(s) = \frac{T(s)}{\max(1, n(s))} \cdot \frac{1}{3600}.$$

The structural sub-map is then

$$\phi_{\text{structure}}(s) = \Big(\bar{\tau}(s),\, N(s)\Big) = \left(\frac{T(s)}{\max(1, n(s)) \cdot 3600},\, \frac{n(s)}{10}\right). \tag{14.3}$$

The normalising constant 10 used in $N(s) = n(s)/10$ is an empirically chosen upper bound for the typical number of segments in a routine session; it matches the factor used in `session.segments.length / 10`. This keeps $N(s)$ in a numerically stable range while still allowing the model to discriminate between fragmented and consolidated sessions.

(3) Interaction dynamics. Let $L(s)$ and $P(s)$ denote the number of laps and pauses in the session, respectively. We define

$$\phi_{\text{interaction}}(s) = \Big(L^*(s),\, P^*(s)\Big) = \left(\frac{L(s)}{10},\, \frac{P(s)}{5}\right). \tag{14.4}$$

From the engine perspective, laps and pauses are discrete events on the same time axis as the segments: each occurrence can be viewed as a delta spike in an underlying counting process. In principle one could model intensities $\lambda_L(s) = L(s)/T(s)$ and $\lambda_P(s) = P(s)/T(s)$; the present implementation uses the simpler scaled counts L^* and P^*, which are sufficient to capture whether a session is unusually fragmented or frequently interrupted.

(4) Content-phase indicators. The final group encodes very coarse information about the *type* of work performed. Let $\text{names}(s)$ denote the multiset of segment names associated with s, written in lowercase. The content-phase sub-map is defined via indicator functions:

$$\phi_{\text{content}}(s) = \big(I_{\text{ther}}(s),\, I_{\text{assess}}(s)\big), \tag{14.5}$$

where

$$I_{\text{ther}}(s) = \mathbf{1}\big\{\exists\, \text{name} \in \text{names}(s) : \text{“therapy”} \subseteq \text{name}\big\},$$
$$I_{\text{assess}}(s) = \mathbf{1}\big\{\exists\, \text{name} \in \text{names}(s) : \text{“assessment”} \subseteq \text{name}\big\}.$$

These binary flags match the logic of `.includes('therapy')` and `.includes('assessment')` in the TypeScript code. They are intentionally minimal: the model does not see raw patient text, diagnostic labels, or medication names; it only sees whether the clinician has marked parts of the session as therapy-like or assessment-like. This supports interpretability (“the model suggested documentation because this was an assessment-heavy morning session”) and aligns with a conservative privacy posture.

Unified feature map. Combining (14.2)–(14.5) gives the eight-dimensional feature map

$$\phi_{\text{sess}}(s) = \big(h(s),\, d(s),\, \bar{\tau}(s),\, L^{*}(s),\, P^{*}(s),\, I_{\text{ther}}(s),\, I_{\text{assess}}(s),\, N(s)\big) \in [0,1]^{6} \times \{0,1\}^{2}, \tag{14.6}$$

which is exactly the vector returned by `extractFeatures` for historical sessions and by the inline feature construction in `predictNextSegment` for live patterns.

By design, all components are either in $[0,1]$ or in the discrete set $\{0,1\}$, and hence

$$\sup_{s \in \mathcal{S}} \|\phi_{\text{sess}}(s)\|_{2} \le \sqrt{8}.$$

This uniform bound simplifies optimisation in the TF.js model and reduces the risk of numerical instabilities on lower-end hardware.

14.1.3 Dataset construction and label space

For learning, SynapseCore treats the successive sessions of a clinician as a sequence

$$(s_1, s_2, \dots, s_N), \qquad s_i \in \mathcal{S}_{\text{hist}},$$

obtained from the browser-local store through `loadHistoricalSessions()`. The feature vectors

$$x_i = \phi_{\text{sess}}(s_i) \in \mathbb{R}^{8}$$

serve as inputs to the model.

The labels are derived from the *first segment* of the subsequent session s_{i+1}, reflecting the design choice in `trainModel` to predict "what comes next in your schedule":

$$z_i = \text{type of first segment in } s_{i+1} \in \{0, \dots, 5\}, \tag{14.7}$$

$$\delta_i = \text{duration of first segment in } s_{i+1} \in [0, \infty). \tag{14.8}$$

The type z_i is produced by the encoder `encodeSegmentName`, then rescaled into $[0,1]$ by dividing by 5; the duration δ_i is expressed in hours:

$$y_i = \left(\frac{z_i}{5},\, \delta_i\right) \in [0,1] \times [0, \infty).$$

Collecting these into matrices

$$X = \begin{bmatrix} x_1^{\top} \\ \vdots \\ x_{N-1}^{\top} \end{bmatrix} \in \mathbb{R}^{(N-1)\times 8}, \qquad Y = \begin{bmatrix} y_1^{\top} \\ \vdots \\ y_{N-1}^{\top} \end{bmatrix} \in \mathbb{R}^{(N-1)\times 2},$$

the TF.js model described in a later subsection learns a function $f_\theta : \mathbb{R}^8 \to \mathbb{R}^2$ by minimising the mean-squared error loss

$$\mathcal{L}(\theta) = \frac{1}{N-1} \sum_{i=1}^{N-1} \big\| f_\theta(x_i) - y_i \big\|_{2}^{2}.$$

At inference time, the same feature map ϕ_{sess} is applied to a live `SessionPattern` object, yielding a feature vector $\tilde{x}$. The model output $f_\theta(\tilde{x}) = (\hat{z}, \hat{\delta})$ is then decoded into a human-readable prediction via

$$\widehat{\text{type}} = \texttt{decodeSegmentType}\big(\hat{z} \cdot 5\big), \tag{14.9}$$

$$\widehat{\text{duration}} = \text{clip}_{[300,7200]}\big(\hat{\delta} \cdot 3600\big), \tag{14.10}$$

where $\text{clip}_{[a,b]}$ truncates to the clinically reasonable range of 5 to 120 minutes, matching the logic in `predictNextSegment`.

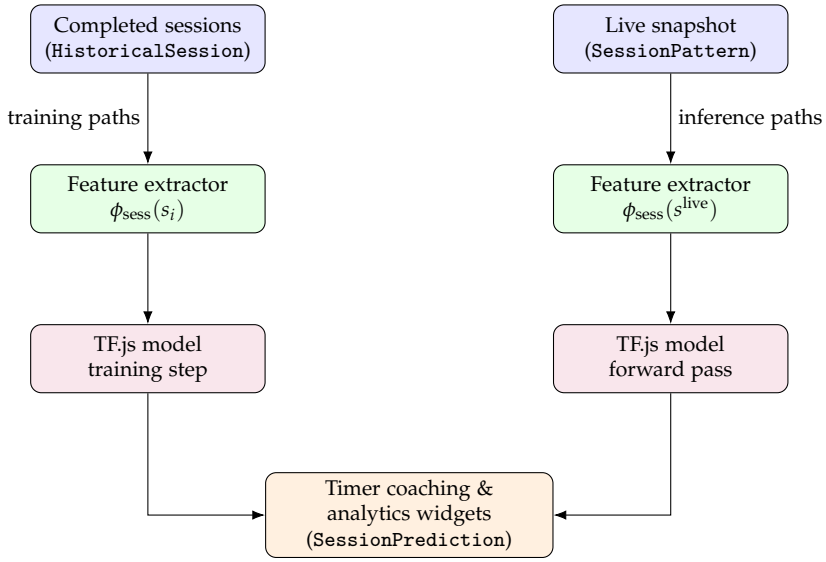


Figure 43: *Two-column vertical schematic of the session feature pipeline. Both training and inference paths use the same feature extractor ϕ_{sess} but differ in whether they update model weights or produce* `SessionPrediction` *objects for the timer UI and analytics widgets.*

14.1.4 Interpretability and clinical reading of features

Table 65 re-organises the components of ϕ_{sess} into vertically arranged categories that mirror how a psychiatrist would informally describe their own schedule. For example, a feature vector with $h \approx 0.3$, $d \approx 0.4$, $\bar{\tau} \approx 0.75$ h, $L^* \approx 0.1$, $P^* \approx 0$, $I_{\text{ther}} = 1$, $I_{\text{assess}} = 0$, $N \approx 0.3$ can be read as "mid-morning, mid-week, few long therapy segments with minimal fragmentation", which aligns directly with everyday clinical language.

This explicit semantic layering is important for safety:

- It guarantees that the model never sees raw clinical narratives, diagnostic codes, or psychometric scores from the MBC engine (Section 6); it only operates on timing and very coarse segment labels.
- It allows the learned behaviour to be inspected at the level of feature importance (e.g. via SHAP in future work), helping to identify spurious correlations such as over-reliance on day-of-week or lap count.

14.1.5 Two-column vertical schematic of the feature pipeline

Figure 43 presents the same feature engineering pipeline as a vertically oriented, two-column schematic, designed to fit safely within the float boundaries of the template. The left column represents the *training* pathway, starting from completed `HistoricalSession` objects and flowing downwards through feature extraction to model updates. The right column represents the *inference* pathway, starting from a live `SessionPattern` and producing a `SessionPrediction` that is consumed by the timer UI and analytics components.

14.2 TensorFlow.js Model Design

While the feature map $\phi_{\text{sess}} : \mathcal{S} \to \mathbb{R}^8$ provides a clinically interpretable representation of each session, the actual learning of patterns and suggestions is delegated to a compact TensorFlow.js model that runs entirely in the browser. This subsection describes the mathematical

form of that model, its implementation in `useSessionML.ts`, and how its outputs are interpreted as predictions about expected duration, segment overrun risk, and suggested timer presets.

From a high-level perspective, the model is a shallow feed-forward network

$$f_\theta : \mathbb{R}^8 \longrightarrow \mathbb{R}^2,$$

where θ denotes the collection of weight matrices and bias vectors. Given an eight-dimensional feature vector $x = \phi_{\text{sess}}(s)$ summarising a historical or live session s, the network produces a two-dimensional output

$$f_\theta(x) = \begin{pmatrix} \hat{z} \\ \hat{\delta} \end{pmatrix},$$

where $\hat{z}$ is a real-valued proxy for the next segment type and $\hat{\delta}$ is a continuous prediction of the next segment's duration (in hours). These two coordinates are then post-processed to obtain (i) a suggested segment preset, (ii) a concrete duration in seconds, and (iii) a derived notion of "overrun risk" when compared with the clinician's current session state.

14.2.1 Network architecture and parameterisation

The concrete architecture implemented in `createModel` is a small *sequential* network with two hidden layers and a two-dimensional linear output layer:

$$\begin{aligned} h_1 &= \sigma\big(W_1 x + b_1\big), \\ h_2 &= \sigma\big(W_2 h_1 + b_2\big), \\ o &= W_3 h_2 + b_3, \end{aligned} \tag{14.11}$$

where

- $x \in \mathbb{R}^8$ is the feature vector discussed in equation (14.6),
- $h_1 \in \mathbb{R}^{16}$ and $h_2 \in \mathbb{R}^8$ are hidden activations,
- $o = (\hat{z}, \hat{\delta})^\top \in \mathbb{R}^2$ is the raw model output,
- $\sigma(\cdot)$ denotes the element-wise rectified linear unit (ReLU),
- $W_1 \in \mathbb{R}^{16 \times 8}$, $W_2 \in \mathbb{R}^{8 \times 16}$, $W_3 \in \mathbb{R}^{2 \times 8}$ are the weight matrices, and
- $b_1 \in \mathbb{R}^{16}$, $b_2 \in \mathbb{R}^8$, $b_3 \in \mathbb{R}^2$ are bias vectors.

In TensorFlow.js this corresponds to:

- a first dense layer with `units: 16`, `inputShape: [8]`, `activation: 'relu'`, and `kernelInitializer: 'heNormal'`,
- a dropout layer with `rate: 0.2`,
- a second dense layer with `units: 8`, `activation: 'relu'`, and
- a final dense layer with `units: 2`, `activation: 'linear'`.

The total number of trainable parameters is modest:

$$\underbrace{16 \cdot 8 + 16}_{W_1, b_1} + \underbrace{8 \cdot 16 + 8}_{W_2, b_2} + \underbrace{2 \cdot 8 + 2}_{W_3, b_3} = 144 + 136 + 18 = 298.$$

This small footprint is deliberate: it ensures that training and inference remain responsive on typical clinical laptops and tablets, and reduces the risk of overfitting given the limited number of sessions likely to be available for any single psychiatrist.

Table 66: *Layer-wise specification of the TensorFlow.js model used for session prediction. The architecture mirrors the implementation in* `useSessionML.ts`*, with a shared backbone for both segment type and duration regression.*

Layer	TensorFlow.js configuration	Clinical / technical role
Input	Shape 1×8 tensor built from $\phi_{\text{sess}}(s)$	Encodes temporal context, structure, interaction dynamics, and content-phase indicators for a single session.
Dense 1	`units: 16, activation: 'relu', kernelInitializer: 'heNormal'`	Learns a non-linear combination of the eight features, emphasising clinically meaningful directions such as "busy mid-week therapy blocks" or "fragmented on-call work".
Dropout	`rate: 0.2` (applied during training only)	Regularises the model by randomly dropping 20% of hidden units, preventing memorisation of a few atypical sessions.
Dense 2	`units: 8, activation: 'relu'`	Compresses the representation into an eight-dimensional bottleneck that feeds both output coordinates, encouraging parameter sharing between segment-type and duration predictions.
Output	`units: 2, activation: 'linear'`	Produces $(\hat{z}, \hat{\delta})$, which are later decoded into a discrete segment label, a concrete duration in seconds, and derived overrun and preset suggestions.

A dropout layer $D_{0.2}$ with rate 0.2 is inserted between the two dense hidden layers. At training time this can be seen as multiplying h_1 component-wise by an i.i.d. Bernoulli mask $m \sim \text{Bernoulli}(0.8)^{16}$ and scaling appropriately, which encourages the network to rely on distributed representations rather than memorising idiosyncratic patterns in a few coordinates:

$$\tilde{h}_1 = D_{0.2}(h_1) = \frac{m \odot h_1}{0.8}, \qquad h_2 = \sigma\big(W_2 \tilde{h}_1 + b_2\big).$$

The model is compiled with the Adam optimiser and mean squared error loss, as in:

```
newModel.compile({
  optimizer: tf.train.adam(0.001),
  loss: 'meanSquaredError',
  metrics: ['mae'],
});
```

Mathematically, given a dataset of $N-1$ training pairs $\{(x_i, y_i)\}_{i=1}^{N-1}$ with $x_i \in \mathbb{R}^8$ and $y_i \in \mathbb{R}^2$ (constructed as described below), the training objective is

$$\mathcal{L}(\theta) = \frac{1}{N-1} \sum_{i=1}^{N-1} \big\| f_\theta(x_i) - y_i \big\|_2^2, \tag{14.12}$$

with the Adam update rule used to approximate gradient descent on this loss.

14.2.2 Dataset construction and multi-task targets

The training data for the model are constructed incrementally from historical sessions stored in the browser. The helper `trainModel(historicalSessions)` first builds feature vectors and labels using the same feature extractor employed at inference time:

$$x_i = \phi_{\text{sess}}(s_i), \qquad y_i = \begin{pmatrix} z_i/5 \\ \delta_i \end{pmatrix}, \tag{14.13}$$

where:

- s_i is the i-th session in chronological order,
- $z_i \in \{0, \ldots, 5\}$ encodes the type of the *first segment* in the *subsequent* session s_{i+1} via `encodeSegmentName`,
- δ_i is the duration (in hours) of that first segment,
- the scaling $z_i/5$ compresses the discrete label into $[0, 1]$ to match the typical range of δ_i.

In TypeScript this corresponds to:

```
const sessionFeatures = extractFeatures(session, new Date(session.timestamp));
features.push(sessionFeatures);

const nextSegmentType = nextSession.segments[0]
  ? encodeSegmentName(nextSession.segments[0].name)
  : 0;
const nextDuration = nextSession.segments[0]
  ? nextSession.segments[0].duration / 3600
  : 0.5;

labels.push([nextSegmentType / 5, nextDuration]);
```

The resulting arrays `features` and `labels` are then turned into tensors via `tf.tensor2d`, yielding matrices $X \in \mathbb{R}^{(N-1)\times 8}$ and $Y \in \mathbb{R}^{(N-1)\times 2}$ as in equation (14.12).

Training uses mini-batch gradient descent with `batchSize: 8`, `epochs: 50`, and `validationSplit: 0.2`, so that 20% of the sessions are held out for validation at each update. An `onEpochEnd` callback updates the `trainingProgress` state, which is surfaced in the `MLInsightsPanel` as a percentage bar and status text (“warming up”, “learning your schedule”, and so on). After training, the model is saved to IndexedDB under the key `'session-predictor'`, enabling subsequent sessions to reuse the trained weights without re-training from scratch.

To avoid uncontrolled growth of the local training set, the helper `saveSessionForTraining` maintains a sliding window of the *most recent* 100 sessions: only these are retained in `'consulton_session_history'` and used for training. This keeps the learning focused on the psychiatrist’s current workflow patterns and reduces the risk that outdated scheduling habits dominate the model.

14.2.3 Inference: expected duration, overrun risk, and presets

At inference time the same feature extractor is applied to the `SessionPattern` describing the current live session:

$$\tilde{x} = \phi_{\text{sess}}(s^{\text{live}}),$$

which is wrapped into a rank-2 tensor of shape 1×8 and fed through the trained model:

$$\begin{pmatrix} \hat{z} \\ \hat{\delta} \end{pmatrix} = f_\theta(\tilde{x}).$$

In `predictNextSegment` this is implemented via `model.predict(input)` followed by `await prediction.array()`.

Expected duration. The second coordinate $\hat{\delta}$ is interpreted as a prediction of segment duration in hours; multiplying by 3600 and rounding yields a duration in seconds:

$$\widehat{D} = \operatorname{round}\big(\hat{\delta} \cdot 3600\big), \tag{14.14}$$

which is then clipped to a clinically reasonable interval $[300, 7200]$ seconds (5–120 minutes) to obtain the suggested timer duration:

$$D_{\text{suggested}} = \min\big\{7200,\, \max\{300, \widehat{D}\}\big\}. \tag{14.15}$$

This quantity is exposed as `suggestedDuration` in the `SessionPrediction` object returned by the hook.

Segment type and suggested preset. The first coordinate $\hat{z}$ is decoded into a discrete segment label via

$$\widehat{k} = \operatorname*{arg\,min}_{k \in \{0,\dots,5\}} \big|k - 5\hat{z}\big|, \qquad \widehat{\text{type}} = \text{decodeSegmentType}(\widehat{k}), \tag{14.16}$$

which mirrors the implementation

```
const [segmentTypeNorm, durationNorm] = predData as number[];
const segmentType = decodeSegmentType(segmentTypeNorm * 5);
```

In the UI, this decoded label is treated as a *preset* suggestion: if $\widehat{\text{type}}$ is `'Therapy'` and $D_{\text{suggested}} \approx 50$ minutes, the timer modal may offer a single-click action to start a "50-minute therapy" segment; if `'Documentation'` and 10–15 minutes, the UI may propose a short documentation preset instead. Formally, one can view this as a mapping

$$g : \mathcal{L} \times \mathbb{R}_{+} \longrightarrow \mathcal{P},$$

where $\mathcal{L}$ is the finite set of segment labels, $\mathbb{R}_{+}$ the space of positive durations, and $\mathcal{P}$ the set of timer presets available in the modal.

Overrun risk and anomaly flags. Although the current implementation exposes a simple boolean flag `isAnomaly` to indicate unusually long sessions (currently defined as those with total duration exceeding four hours), the same architecture naturally supports a smoother notion of *overrun risk*. Let R_{rem} denote the remaining time in the current session schedule (for example, the remaining slot in the clinician's calendar) and $D_{\text{suggested}}$ the expected duration from equation (14.14). A simple risk score is

$$\rho = \frac{D_{\text{suggested}}}{R_{\text{rem}} + \varepsilon},$$

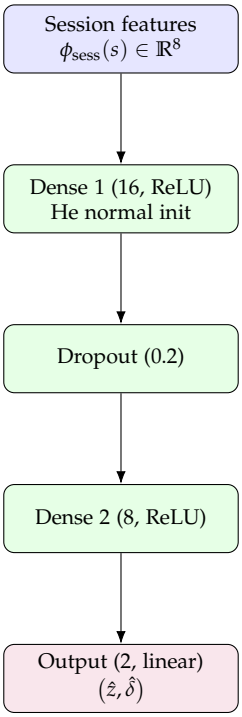


Figure 44: *Vertically organised schematic of the TensorFlow.js model used for session prediction. An eight-dimensional feature vector is processed by two ReLU-activated dense layers with dropout regularisation, followed by a linear two-dimensional output layer encoding segment type and expected duration.*

with a small $\varepsilon > 0$ to avoid division by zero. A probability-like overrun measure can then be obtained via the logistic transform

$$p_{\text{overrun}} = \sigma_{\beta,\tau}(\rho) = \frac{1}{1 + \exp\big(-\beta(\rho - \tau)\big)}, \tag{14.17}$$

with slope parameter $\beta > 0$ and threshold τ chosen so that $\rho = \tau$ corresponds to a 50% overrun risk.

In a future iteration, p_{overrun} can be calculated from the existing model outputs without re-training, and surfaced alongside the boolean anomaly flag. This would allow the timer coaching panel to distinguish between mild overruns (e.g. 60% risk, suggesting gentle pacing prompts) and critical ones (e.g. $p_{\text{overrun}} > 0.9$, suggesting more directive interventions or schedule adjustments).

14.3 Data Pipeline

The TensorFlow.js model described in Section 14.2 is only clinically meaningful if it is trained and updated on a representative stream of the psychiatrist's own sessions. SynapseCore therefore implements a lightweight *data pipeline* that captures completed sessions from the timer engine, stores them locally in a bounded, privacy-preserving history, and periodically refreshes the session model when sufficient new data have accumulated.

Operationally this pipeline is implemented entirely in the browser via the functions `loadHistoricalSessions()` and `saveSessionForTraining()`, together with the internal `trainModel` helper and the IndexedDB-backed model storage in `useSessionML`. Mathematically, the pipeline can be understood as a composition of simple maps on finite sequences of session objects and model parameters. This subsection formalises that composition, describes

the concrete implementations, and explains why we restrict ourselves to the N most recent sessions for any training run.

14.3.1 Local session store and robust retrieval

At the storage layer, the pipeline treats the browser's `localStorage` as a persistent key–value store for de-identified timer sessions. All historical training data are kept under a single key, `'consulton_session_history'`, whose value is a JSON-encoded array of `HistoricalSession` objects.

From a mathematical perspective, we can model the contents of this key as a finite sequence

$$H = (s_1, s_2, \ldots, s_M) \in \mathcal{S}_{\text{hist}}^{*},$$

where $\mathcal{S}_{\text{hist}}$ is the space of completed timer trajectories described in Section 14.1 and M is the number of stored sessions. The retrieval function `loadHistoricalSessions()` implements a robust map

$$L : \{\texttt{localStorage}\} \longrightarrow \mathcal{S}_{\text{hist}}^{*}, \tag{14.18}$$

which decodes the JSON value if present and well formed, and otherwise returns an empty sequence:

$$L(\cdot) = \begin{cases} (s_1, \ldots, s_M), & \text{if the key } \texttt{'consulton\_session\_history'} \text{ exists and parses,} \\ \varnothing, & \text{if the key is missing or JSON decoding fails.} \end{cases} \tag{14.19}$$

In TypeScript this is realised as:

```
function loadHistoricalSessions(): HistoricalSession[] {
  try {
    const stored = localStorage.getItem('consulton_session_history');
    if (!stored) return [];
    return JSON.parse(stored) as HistoricalSession[];
  } catch {
    return [];
  }
}
```

The try/catch block is crucial: it ensures that any corruption of the stored JSON (for example, due to manual editing or browser extensions) does not break the timer or ML components. Instead, the pipeline gracefully falls back to an empty history, in which case the downstream `trainModel` routine simply reports that there are not yet enough sessions to learn from.

14.3.2 Sliding window and recency operator

To avoid unbounded growth of the local history and to focus the model on the clinician's current practice style, SynapseCore restricts the training set to the most recent N sessions, with $N = 100$ in the current implementation. Formally, let

$$H = (s_1, \ldots, s_M), \qquad M \geq 0,$$

be the sequence returned by L in equation (14.18). The *recency operator* $R_N : \mathcal{S}^*_{\text{hist}} \to \mathcal{S}^*_{\text{hist}}$ is defined by

$$R_N(H) = \begin{cases} H, & \text{if } M \le N, \\ (s_{M-N+1}, \ldots, s_M), & \text{if } M > N, \end{cases} \tag{14.20}$$

i.e. it keeps the last N sessions and discards the rest. In JavaScript this is just the familiar `slice(-100)` pattern:

```
const sessions = loadHistoricalSessions();
sessions.push(sessionData);

const recentSessions = sessions.slice(-100);
localStorage.setItem(
  'consulton_session_history',
  JSON.stringify(recentSessions)
);
```

Two properties of R_N are worth emphasising:

- It is *idempotent*: $R_N(R_N(H)) = R_N(H)$ for all histories H. This means that applying the retention rule repeatedly does not gradually erode the last N sessions; they are preserved as long as no new session arrives.
- It ensures a uniform bound on storage and computational cost: for fixed N, both the memory footprint and the time required to compute features and loss scales linearly with N, independent of how long the psychiatrist has been using the tool.

Typical browser `localStorage` implementations offer only a few megabytes of storage; by bounding N we can guarantee that the JSON representation of the history remains well below that limit even when each `HistoricalSession` object contains dozens of segments and fine-grained lap information.

14.3.3 Save-and-train behaviour

The function `saveSessionForTraining()` is the main entry point from the timer engine into the ML pipeline. Whenever a session is completed and summarised into a `HistoricalSession`, the timer subsystem calls this hook, which performs three steps:

1. *Extend the history.* A new session s_{new} is appended to the existing sequence $H = L(\cdot)$.
2. *Apply the recency rule.* The extended sequence $H' = (H, s_{\text{new}})$ is truncated via R_N to obtain $H'' = R_N(H')$, which is written back to `localStorage`.
3. *Trigger training when appropriate.* If the length of H'' is at least a minimal threshold $M_{\min} = 10$ and the model is not currently in a training phase, the asynchronous `trainModel` routine is invoked with H'' as its data source.

In code this is expressed as:

```
const saveSessionForTraining = React.useCallback(
  async (sessionData: HistoricalSession) => {
    try {
      const sessions = loadHistoricalSessions();
      sessions.push(sessionData);

      const recentSessions = sessions.slice(-100);
```

```
      localStorage.setItem(
        'consulton_session_history',
        JSON.stringify(recentSessions)
      );

      if (recentSessions.length >= 10 && !isTraining) {
        await trainModel(recentSessions);
      }
    } catch (error) {
      console.error('× Failed to save session:', error);
    }
  },
  [isTraining, trainModel]
);
```

Abstracting away React-specific details, we can formalise the pipeline step for a single new session as the composite map

$$\Psi(s_{\text{new}}) = R_N\big(L(\cdot) \,\|\, s_{\text{new}}\big), \tag{14.21}$$

where $\|$ denotes sequence concatenation. Training is then conditionally triggered whenever $|\Psi(s_{\text{new}})| \geq M_{\min}$ and no training process is already active. The threshold $M_{\min} = 10$ reflects an engineering trade-off: we need enough data for the model to capture meaningful variation in session patterns, but training on very few sessions risks severe overfitting and unstable suggestions.

14.3.4 Model persistence and symmetry of read/write paths

A second axis of the data pipeline concerns the storage of the model *parameters* themselves. Whereas the session history resides in `localStorage`, the TensorFlow.js model is persisted in IndexedDB under the key `'session-predictor'`. This separation of concerns has two advantages:

- localStorage is optimised for small, synchronous key–value reads; it is ideal for JSON-encoded histories used by the training pipeline.
- IndexedDB is better suited for binary blobs and larger objects such as TF.js model weights; it supports streaming and atomic replacement of model artefacts.

Within `useSessionML`, a `React.useEffect` hook attempts to load the model from IndexedDB when the timer panel is first mounted:

```
React.useEffect(() => {
  async function initModel() {
    try {
      const loadedModel = await tf.loadLayersModel(
        'indexeddb://session-predictor'
      );
      setModel(loadedModel);
    } catch {
      const newModel = createModel();
      setModel(newModel);
    }
  }
```

Table 67: *Key components of the session ML data pipeline, from timer engine output to persisted model weights. All operations are performed client-side within the browser and rely on bounded histories of the most recent $N = 100$ sessions.*

Component	Role in the pipeline
`localStorage` key `'consulton_session_history'`	Holds a JSON-encoded array of `HistoricalSession` objects, representing the finite history H_t used for training. Updated only via `saveSessionForTraining` to enforce the recency rule R_N.
`loadHistoricalSessions()`	Robust reader for the session history. Returns an empty array in the presence of missing keys or parse errors, ensuring that the timer and ML layers degrade gracefully rather than failing hard.
`saveSessionForTraining(session)`	Appends a new completed session to the history, applies the recency operator R_N by keeping only the last $N = 100$ sessions, persists the result to `localStorage`, and conditionally triggers `trainModel` when at least $M_{\min} = 10$ sessions are available.
`trainModel(sessions)`	Constructs feature and label tensors from the provided sessions, fits the TensorFlow.js model using mini-batch Adam, updates training progress indicators, and saves the resulting weights to IndexedDB under `'session-predictor'`.
IndexedDB key `'session-predictor'`	Stores the most recent version of the TF.js model parameters θ_t. Loaded by `initModel` on startup and overwritten when a new training run finishes.
`useSessionML` hook	Exposes a cohesive API to the rest of the application: `predictNextSegment` for inference, `saveSessionForTraining` for ingestion, and the flags `isTraining` and `trainingProgress` for UI feedback in the ML insights panel.

```
  initModel();
}, []);
```

If a previously trained model is found, it is restored; otherwise a fresh model is created via `createModel()` as described in Section 14.2. After each successful call to `trainModel`, the updated weights are saved back to IndexedDB using `model.save('indexeddb://session-predictor')`, thereby closing the loop between data ingestion, training, and deployment.

Viewed abstractly, the combined data and model persistence can be described by two coupled state variables:

$$H_t = R_N(H_{t-1} \,\|\, s_t), \qquad \theta_t = \begin{cases} \theta_{t-1}, & \text{if no training is triggered at time } t, \\ \mathrm{Train}\big(\theta_{t-1}, H_t\big), & \text{if } |H_t| \geq M_{\min}, \end{cases}$$

where θ_t are the model parameters at update step t and Train denotes the optimisation procedure in equation (14.12). The functions `loadHistoricalSessions()`, `saveSessionForTraining()`, and `trainModel` together implement this discrete-time dynamical system, with state stored across browser restarts via localStorage and IndexedDB.

14.3.5 Vertically organised data-flow schematic

Figure 45 offers a vertically oriented, colour-coded view of the data pipeline, constrained to a single column so that it remains comfortably within the page margins. The diagram highlights the asymmetry between *data* persistence in `localStorage` and *model* persistence in IndexedDB, while emphasising that all processing—from ingestion to training and inference—occurs locally on the clinician's machine.

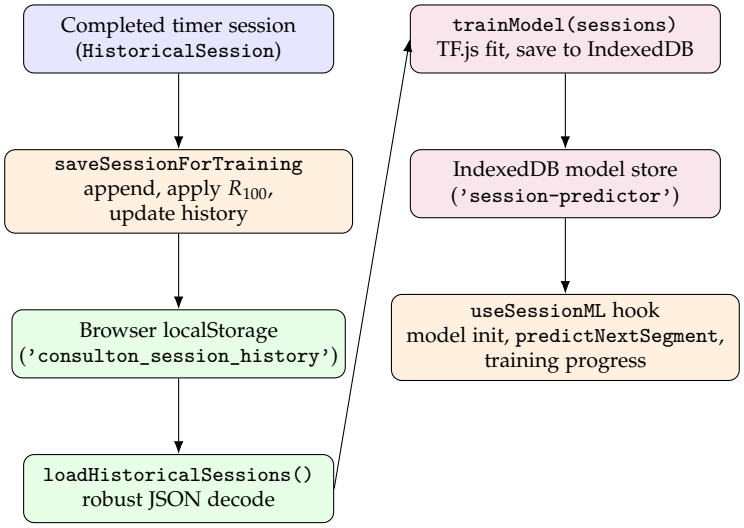


Figure 45: *Two-column parallel schematic of the session ML data pipeline. The left lane covers timer-session ingestion and historical persistence, while the right lane tracks model training, storage, and runtime inference handled by the* `useSessionML` *hook.*

14.4 Potential Clinical Metrics

The combination of timer traces, feature engineering (Section 14.1), and the TensorFlow.js model (Section 14.2) yields a rich stream of structured, de-identified data about how sessions unfold in real time. Beyond providing point predictions for upcoming segment durations, this infrastructure makes it possible to define clinically interpretable *metrics* that quantify workload, pacing, and schedule strain in a way that is consistent with the broader measurement-based care (MBC) philosophy of SynapseCore.

Crucially, all metrics introduced in this subsection depend only on timer engine outputs and model predictions; no diagnostic codes, free text, or psychometric scores are required. They can therefore be computed safely in the browser and, if desired, surfaced as unobtrusive overlays in the timer UI, weekly dashboards, or supervision reports. We group these metrics into four broad families: (i) prediction calibration indices, (ii) overload and capacity strain indices, (iii) within-session pacing metrics, and (iv) longitudinal stability and variability measures.

14.4.1 Prediction calibration: per-segment and per-session

Let a completed session s consist of n segments with actual durations $(A_1, \ldots, A_n)$ in seconds, and let $(\widehat{D}_1, \ldots, \widehat{D}_n)$ be the corresponding predicted durations produced by the model at the time each segment was initiated (derived from equation (14.14)). We define for each segment $j \in \{1, \ldots, n\}$:

$$e_j = A_j - \widehat{D}_j, \qquad r_j = \frac{A_j - \widehat{D}_j}{\max\{1, \widehat{D}_j\}}, \tag{14.22}$$

where e_j is the absolute error (in seconds) and r_j is the dimensionless relative error. The `SessionPrediction` object could, in a future extension, store these values as part of a non-identifying analytics payload.

Aggregating across segments yields session-level calibration indices:

$$\mathrm{MAE}_{\mathrm{sess}} = \frac{1}{n}\sum_{j=1}^{n}|e_j|, \tag{14.23}$$

$$\mathrm{RMSE}_{\mathrm{sess}} = \sqrt{\frac{1}{n}\sum_{j=1}^{n}e_j^2}, \tag{14.24}$$

$$\mathrm{Bias}_{\mathrm{sess}} = \frac{1}{n}\sum_{j=1}^{n}e_j. \tag{14.25}$$

Here, $\mathrm{MAE}_{\mathrm{sess}}$ and $\mathrm{RMSE}_{\mathrm{sess}}$ characterise overall accuracy for that session, while $\mathrm{Bias}_{\mathrm{sess}}$ indicates whether the model tends to systematically over- or under-estimate durations for that particular clinician, clinic, or patient.

Over a collection of K sessions, one can similarly compute a global calibration profile:

$$\mathrm{MAE}_{\mathrm{global}} = \frac{\sum_{i=1}^{K}\sum_{j=1}^{n_i}|e_{i,j}|}{\sum_{i=1}^{K}n_i}, \qquad \mathrm{Bias}_{\mathrm{global}} = \frac{\sum_{i=1}^{K}\sum_{j=1}^{n_i}e_{i,j}}{\sum_{i=1}^{K}n_i}, \tag{14.26}$$

where n_i is the number of segments in session i and $e_{i,j}$ is the error for segment j in that session. These global calibration metrics can be displayed in an *ML insights* panel next to the timer, conveying to the psychiatrist how well the system has adapted to their current practice style.

Because all quantities are computed client-side from `HistoricalSession` objects and `SessionPrediction` snapshots, no central storage is required; a sliding window of recent sessions (maintained by the data pipeline in Section 14.3) is sufficient to keep the metrics up to date.

14.4.2 Overload and capacity strain indices

Psychiatrists rarely work in an abstract time continuum; clinical work is distributed across fixed calendar slots with limited flexibility. The model's predicted duration and the timer's realised duration can therefore be combined to form indices of *overload* and *capacity strain* at both session and day levels.

Session-level overload. Let $S > 0$ denote the scheduled slot for a given appointment (e.g. $S = 50$ minutes $= 3000$ seconds for a standard therapy hour). The actual and predicted total durations are

$$T^{\mathrm{act}} = \sum_{j=1}^{n}A_j, \qquad T^{\mathrm{pred}} = \sum_{j=1}^{n}\widehat{D}_j.$$

The *retrospective* overload index is

$$\mathrm{OI}^{\mathrm{retro}} = \frac{T^{\mathrm{act}} - S}{S}, \tag{14.27}$$

while the *prospective* overload index is

$$\mathrm{OI}^{\mathrm{pros}} = \frac{T^{\mathrm{pred}} - S}{S}, \tag{14.28}$$

mirroring the definitions introduced earlier. The calibration gap

$$\Delta_{\mathrm{OI}} = \mathrm{OI}^{\mathrm{retro}} - \mathrm{OI}^{\mathrm{pros}} \tag{14.29}$$

indicates whether the model (and, by extension, the presets) are realistic for that appointment type.

Day-level strain. For a given day d with sessions $s^{(1)}, \dots, s^{(m)}$ and slots $S^{(1)}, \dots, S^{(m)}$, we can define aggregate overload indices:

$$\mathrm{OI}_{\mathrm{day}}^{\mathrm{retro}} = \frac{\sum_{k=1}^{m} T_{(k)}^{\mathrm{act}} - \sum_{k=1}^{m} S^{(k)}}{\sum_{k=1}^{m} S^{(k)}}, \tag{14.30}$$

$$\mathrm{OI}_{\mathrm{day}}^{\mathrm{pros}} = \frac{\sum_{k=1}^{m} T_{(k)}^{\mathrm{pred}} - \sum_{k=1}^{m} S^{(k)}}{\sum_{k=1}^{m} S^{(k)}}. \tag{14.31}$$

These quantities summarise, in a single pair of numbers, the extent to which a day's planned and realised workload exceeded available capacity. When combined with calendar metadata (e.g. on-call vs routine clinic) they can be used to design more humane rota patterns or to spot excessively overloaded sequences of days.

Overrun probability proxies. In the absence of an explicit probabilistic model, the logistic transform introduced in equation (14.17) can be applied to either session-level or day-level overload ratios to produce probability-like scores. For example, at the beginning of a session, one can compute

$$\rho = \frac{T^{\mathrm{pred}}}{S}, \qquad p_{\mathrm{overrun}} = \sigma_{\beta,\tau}(\rho),$$

with slope β and threshold τ chosen so that $\rho = \tau$ corresponds to a neutral "coin-flip" risk of overrun. This p_{overrun} can feed directly into colour-coded highlights in the timer modal (e.g. green for low, amber for moderate, red for high risk), without exposing any additional technical detail to the clinician.

14.4.3 Within-session pacing and drift metrics

Even when overall overload is manageable, the *distribution* of time within a session matters. Clinicians routinely report that they end up compressing therapeutic work or documentation into the final minutes, especially when assessment segments run long. The cumulative trajectories

$$C_j^{\mathrm{act}} = \sum_{k=1}^{j} A_k, \qquad C_j^{\mathrm{pred}} = \sum_{k=1}^{j} \widehat{D}_k$$

provide a natural basis for quantifying such drift.

Shape-normalised pacing curves. To control for total session length, we normalise cumulative time by the respective totals:

$$u_j = \frac{C_j^{\mathrm{act}}}{T^{\mathrm{act}}}, \qquad v_j = \frac{C_j^{\mathrm{pred}}}{T^{\mathrm{pred}}}, \qquad j = 1, \dots, n.$$

Both (u_j) and (v_j) are increasing sequences in $[0, 1]$ with $u_n = v_n = 1$.

Pacing deviation index. The *pacing deviation index* is the discrete L^1 distance between the two curves, as introduced in equation (14.32):

$$\mathrm{PDI} = \frac{1}{n} \sum_{j=1}^{n} \left| u_j - v_j \right|. \tag{14.32}$$

Small values of PDI indicate that the clinician's actual time usage closely follows the model-implied pacing; large values reflect significant drift, often experienced subjectively as "rushing" or "running behind".

Segment-concentration index. A complementary summary is the concentration of time across segments. Let $p_j = A_j / T^{\text{act}}$ be the proportion of session time spent in segment j. We can define a Gini-like concentration index

$$\text{SCI} = \frac{\sum_{j=1}^{n} \sum_{k=1}^{n} |p_j - p_k|}{2n \sum_{j=1}^{n} p_j}, \tag{14.33}$$

which lies in $[0, 1]$ and becomes large when most of the time is concentrated in a small number of segments. High values of SCI for assessment segments, for example, signal that assessment is dominating the session at the expense of other work.

By computing SCI both for actual and predicted p_j, one can assess whether the psychiatrist consistently drifts towards more concentrated (or more fragmented) patterns than they originally anticipated when selecting presets.

Content-specific pacing ratios. Using the content-phase indicators from equation (14.5), we can define separate pacing ratios for assessment, therapy, and documentation. For a given phase label $c \in \{\text{assess}, \text{ther}, \text{doc}\}$, let $w_j^{(c)}$ be 1 if segment j belongs to that phase and 0 otherwise. Fix a threshold $\alpha \in (0, 1)$ and define

$$J_{\text{early}}(\alpha) = \left\{ j : C_j^{\text{act}} \le \alpha T^{\text{act}} \right\}, \qquad J_{\text{late}}(\alpha) = \left\{ j : C_j^{\text{act}} > \alpha T^{\text{act}} \right\}.$$

The early–late ratio for phase c is then

$$\rho_c(\alpha) = \frac{\sum_{j \in J_{\text{early}}(\alpha)} w_j^{(c)} A_j}{\sum_{j \in J_{\text{late}}(\alpha)} w_j^{(c)} A_j + \varepsilon}, \tag{14.34}$$

with a small $\varepsilon > 0$ for numerical stability. For $c = \text{ther}$, $\rho_{\text{ther}}(\alpha) > 1$ indicates front-loaded therapy work; for $c = \text{doc}$, $\rho_{\text{doc}}(\alpha) < 1$ suggests that documentation is predominantly back-loaded into the end of the session.

Comparing $\rho_c(\alpha)$ to the analogous ratio computed from predicted durations (using $\widehat{D}_j$ instead of A_j) reveals systematic discrepancies between intended and realised pacing for different phases of the consultation.

14.4.4 Longitudinal stability and variability

The metrics introduced above can be computed for each session and then aggregated over time to form longitudinal profiles. Let $\mathcal{M}(s)$ be any of the scalar metrics discussed (e.g. OI^{retro}, PDI, $\rho_{\text{ther}}(\alpha)$). Over a window of W sessions, we can define rolling averages and variances:

$$\overline{\mathcal{M}}_t = \frac{1}{W} \sum_{i=t-W+1}^{t} \mathcal{M}\left(s^{(i)}\right), \tag{14.35}$$

$$\text{Var}_t(\mathcal{M}) = \frac{1}{W} \sum_{i=t-W+1}^{t} \left(\mathcal{M}\left(s^{(i)}\right) - \overline{\mathcal{M}}_t \right)^2, \tag{14.36}$$

Table 68: *Expanded overview of potential clinical metrics derived from predicted vs actual durations. All metrics can be computed from timer and session-ML outputs without accessing narrative or diagnostic content.*

Metric	Formal definition	Clinical interpretation
Session MAE, RMSE, bias	$\mathrm{MAE}_{\mathrm{sess}}$, $\mathrm{RMSE}_{\mathrm{sess}}$, $\mathrm{Bias}_{\mathrm{sess}}$ (equations (14.23)–(14.25))	Accuracy and directional bias of duration predictions within a single session; high values suggest mis-calibrated presets or changes in practice style not yet absorbed by the model.
Retrospective and prospective overload	$\mathrm{OI}^{\mathrm{retro}}$, $\mathrm{OI}^{\mathrm{pros}}$ (equations (14.27)–(14.28))	Degree to which sessions overshoot or undershoot their scheduled slots, both as experienced (*retro*) and as anticipated by the model (*pros*).
Overload calibration gap	$\Delta_{\mathrm{OI}} = \mathrm{OI}^{\mathrm{retro}} - \mathrm{OI}^{\mathrm{pros}}$ (equation (14.29))	Alignment between predicted and realised strain; systematic discrepancies may motivate retraining, preset adjustments, or schedule redesign.
Pacing deviation index	$\mathrm{PDI} = \frac{1}{n}\sum_{j=1}^{n} \lvert u_j - v_j \rvert$ (equation (14.32))	Within-session drift from model-implied pacing; large values indicate that work is being compressed or delayed relative to the intended structure.
Segment concentration index	$\mathrm{SCI} = \dfrac{\sum_{j,k} \lvert p_j - p_k \rvert}{2n \sum_j p_j}$ (see text)	Degree to which session time is concentrated in a small number of segments; high values for assessment or documentation may signal imbalance in the consultation.
Phase-specific pacing ratios	$\rho_c(\alpha) = \dfrac{\sum_{j \in J_{\mathrm{early}}(\alpha)} w_j^{(c)} A_j}{\sum_{j \in J_{\mathrm{late}}(\alpha)} w_j^{(c)} A_j + \varepsilon}$ (equation (14.34))	Balance of assessment, therapy, or documentation time between early and late phases of the session; supports reflective pacing adjustments in supervision or self-monitoring.

for $t \geq W$. These rolling statistics capture both the *central tendency* of the clinician's behaviour (e.g. typical overload) and the *stability* of that behaviour over weeks.

In principle, these longitudinal summaries can be correlated with other elements of the SynapseCore ecosystem (e.g. MBC scores, risk flow usage frequency) to support service-level quality improvement or self-supervision. Because the metrics are derived from timing and content-phase labels rather than from patient-level data, such correlations can be explored without compromising patient confidentiality.

14.4.5 Metric computation pipeline

Figure 46 summarises the computation flow for these metrics as a vertically organised, single-column schematic designed to fit within the page layout.

Timer traces and model predictions are first combined into a generic prediction–error layer, from which overload, pacing, and concentration indices are derived. These indices can

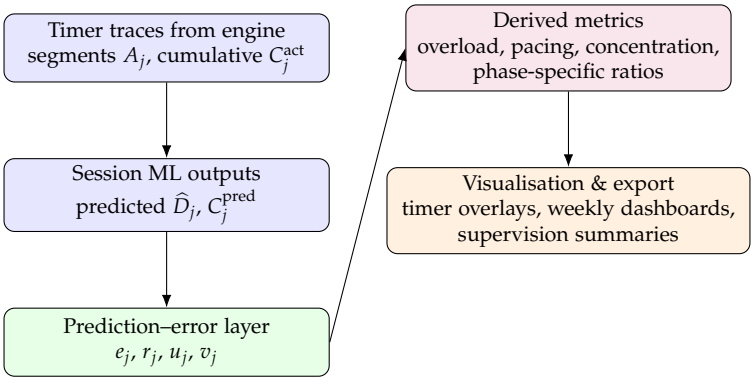


Figure 46: *Two-column parallel version of the session-metrics pipeline. The left stream processes timer-derived segments and session-ML predictions, and the right stream computes derived metrics and exports visual summaries.*

then be aggregated longitudinally and rendered in compact visualisations such as weekly overload graphs, pacing heatmaps, or supervision-ready summaries.

Chapter IV: Evaluation, Governance, and Implications

Sections 15–19

Overview. Part IV shifts the focus from construction to evaluation, governance, and implications for clinical practice. Section 15 outlines the evaluation strategy, covering technical benchmarks (latency, token use, timer drift), usability studies, clinical content validation, and safety and error analysis. Section 16 discusses ethical, legal, and regulatory considerations, including intended use, privacy and security, data governance, and alignment with emerging health-care AI regulation. Sections 17 and 18 articulate limitations and future work, such as EHR and FHIR integration, advanced risk models, team-based workflows, and domain generalisation. Section 19 closes the manuscript with a synthesis of the architecture and its implications for digital psychiatry.

If we denote by $\mathbf{C} = \{C_1, \ldots, C_6\}$ the set of core contributions and by E the evaluation and governance space (benchmarks, studies, ethical and regulatory constraints), Part IV can be viewed as analysing the mapping

$$\mathcal{E} : \mathbf{C} \longrightarrow E,$$

linking the technical artefacts developed in Parts II and III to their clinical, organisational, and regulatory consequences.

15 Evaluation Strategy

The SynapseCore workbench is intended to function both as a day-to-day clinical companion and as a programmable research environment. Any such system must therefore be evaluated along multiple, interacting axes: raw technical performance, subjective usability, clinical content fidelity, and safety under realistic workloads. The present section describes the overarching evaluation strategy and, in Section 15.1, formalises the technical benchmarking layer that underpins the other dimensions.

From a technical standpoint, the workbench exposes a rich telemetry surface (Section 12) via OpenTelemetry-compatible spans and counters, together with internal event streams from the AI orchestration engine (Section 4), the session timer (Section 14), and the flows and tools subsystems. The evaluation strategy deliberately builds on this instrumentation instead of introducing an external benchmarking harness that would risk measuring an artificial code path. Every benchmark is framed as a particular pattern of normal usage—for example, running a 50-minute therapy session with timer laps, completing an agitation flow, or creating a consult letter with multiple AI routes—and is then quantified using the existing observability hooks.

Conceptually, the evaluation plan can be described as a four-layer stack:

1. *Technical benchmarks (Section 15.1):* latency, throughput, timer accuracy, and resource usage under controlled but realistic scenarios.
2. *Usability studies:* structured observation of clinicians using SynapseCore for canonical tasks, combined with standard usability instruments and qualitative feedback.
3. *Clinical content validation:* expert review of scale calculators, flows, and AI-generated narratives against current guidelines and measurement-based care conventions.
4. *Safety and error analysis:* targeted review of AI outputs and audit logs to identify hallucinations, unsafe suggestions, or failure modes in high-risk workflows such as risk/safety assessments.

The remainder of this section elaborates these layers. The present subsection focuses on the first layer and provides concrete definitions for the latency, drift, and resource metrics used to characterise the system under load. Subsequent subsections (to be developed) will build on these primitives to describe usability protocols, clinical validation methods, and safety audits in a manner suitable for replication in independent deployments.

15.1 Technical Benchmarks

The technical benchmarking layer has three primary goals:

- to characterise the latency and throughput of each AI route (provider–model combination) under representative prompt workloads;
- to quantify timer accuracy and drift for both stopwatch and countdown modes over clinically relevant durations; and
- to monitor memory and CPU behaviour of the flows, timer, and IDE modules during extended sessions.

All three goals are implemented using the same underlying mechanisms: OpenTelemetry spans and histograms defined in `src/observability/otel.ts`, wrapped helper functions such as `withSpan` in `src/observability/spans.ts`, and lightweight counters emitted from the AI orchestration layer, timer hooks, and tools exports. This subsection describes the benchmark design in a provider-agnostic, system-level fashion.

15.1.1 Latency and Throughput for AI Routes

Let $\mathcal{R}$ denote the finite set of AI routes exposed to the user, where each route $r \in \mathcal{R}$ is a combination of *provider*, *model*, and *sampling preset* as defined in the AI configuration store (Section 4). For each route r we observe a stream of requests indexed by $i = 1, \dots, n_r$, and record:

- the end-to-end latency $\ell_{r,i} \in \mathbb{R}_{\geq 0}$, measured in milliseconds from the time the user submits a prompt in the SynapseCore panel to the time the stream is marked complete;
- the number of prompt tokens $p_{r,i}$ and completion tokens $c_{r,i}$, as recorded by the counters `tokens_prompt` and `tokens_completion` in the `METRICS` object; and
- the estimated cost $k_{r,i}$ in USD, as recorded by the `cost_usd` counter (where the estimation function is model-specific but deterministic given $(p_{r,i}, c_{r,i})$).

For each $r \in \mathcal{R}$ we form empirical summaries

$$\hat{\mu}_r^{(\ell)} = \frac{1}{n_r} \sum_{i=1}^{n_r} \ell_{r,i}, \qquad \hat{\sigma}_r^{(\ell)} = \sqrt{\frac{1}{n_r - 1} \sum_{i=1}^{n_r} \left(\ell_{r,i} - \hat{\mu}_r^{(\ell)}\right)^2}, \tag{15.1}$$

$$\hat{\mu}_r^{(p)} = \frac{1}{n_r} \sum_{i=1}^{n_r} p_{r,i}, \qquad \hat{\mu}_r^{(c)} = \frac{1}{n_r} \sum_{i=1}^{n_r} c_{r,i}, \tag{15.2}$$

and estimate quantiles such as the empirical 95th percentile $\hat{q}_{0.95}^{(\ell,r)}$ of the latency distribution by standard order-statistic methods. These values are used to define *performance profiles* for each route, for example:

$$\pi(r) = \left(\hat{\mu}_r^{(\ell)}, \hat{q}_{0.95}^{(\ell,r)}, \hat{\mu}_r^{(p)}, \hat{\mu}_r^{(c)}\right),$$

and to compute cost-normalised metrics such as latency per thousand tokens:

$$\lambda_r = \frac{\hat{\mu}_r^{(\ell)}}{\hat{\mu}_r^{(p)} + \hat{\mu}_r^{(c)}} \times 10^3 \quad \text{[ms per 1k tokens]}.$$

The instrumentation for these quantities is already present in the observability layer: every AI request is wrapped in a `withSpan` call that times the request latency and records it in the `req_latency_ms` histogram, while token counts and cost estimates are updated via the respective counters. The evaluation strategy does not require additional code in the core path; instead, it defines:

1. a deterministic set of benchmarking scenarios (for example, a standardised intake prompt, a risk summary prompt, and a multi-turn session recap) and
2. a small set of route-level summary statistics $(\hat{\mu}_r^{(\ell)}, \hat{q}_{0.95}^{(\ell,r)}, \lambda_r)$ that can be computed from exported telemetry.

This design allows the same pipeline to support both offline benchmarking (e.g. comparing providers on a fixed prompt suite) and continuous monitoring in real use. In a clinical deployment, an administrator can define thresholds such as $\hat{q}_{0.95}^{(\ell,r)} < 2.5\,\text{s}$ and $\lambda_r < 1.5\,\text{s}$ per thousand tokens, and configure alerts or route de-prioritisation when these properties are violated.

15.1.2 Timer Accuracy and Drift

The session timer (Section 14) serves as a time-keeping spine for many other subsystems: laps are used to anchor clinical events, countdowns are used to manage segment transitions, and

timer state feeds into future session-level learning components. For this reason, systematic timer error is unacceptable. The evaluation strategy therefore defines explicit *drift benchmarks*.

Let $T_{\mathrm{ref}}(t)$ denote a high-resolution reference clock, for example, the browser's `performance.now()` API. Let $T_{\mathrm{syn}}(t)$ denote the elapsed time reported by the timer engine when run in stopwatch mode for a target duration $t \in \{5, 15, 30, 50\}$ minutes. For each target duration t we perform M independent runs and compute the drift

$$\Delta_{t,j} = T_{\mathrm{syn}}^{(j)}(t) - T_{\mathrm{ref}}^{(j)}(t), \quad j = 1, \ldots, M.$$

We then summarise the drift by its mean and maximum absolute value,

$$\hat{\mu}_t^{(\Delta)} = \frac{1}{M} \sum_{j=1}^{M} \Delta_{t,j}, \qquad \hat{\delta}_t^{\max} = \max_{1 \le j \le M} \left|\Delta_{t,j}\right|. \tag{15.3}$$

The acceptance criteria are formulated in clinical terms: for example, one can require that for all $t \le 50$ minutes we have $|\hat{\mu}_t^{(\Delta)}| < 250\,\mathrm{ms}$ and $\hat{\delta}_t^{\max} < 1\,\mathrm{s}$ across a representative mix of browsers and hardware profiles. In countdown mode, analogous experiments are performed by setting a countdown of length t and recording the difference between the moment when the timer reaches zero and the moment when the reference clock indicates t has elapsed.

Although these experiments can be run manually using browser Developer Tools, the SynapseCore design allows them to be automated: timer state updates are already driven by a discrete *tick* function in `src/centerpanel/components/timerEngine.ts`, and test harnesses can simulate ticks driven by a synthetic reference clock rather than real time. The evaluation methodology thus consists of:

1. defining a standard suite of stopwatch and countdown scenarios;
2. executing them in both real browsers and simulated environments where the tick schedule is controlled; and
3. computing $(\hat{\mu}_t^{(\Delta)}, \hat{\delta}_t^{\max})$ and verifying that they remain within predefined clinical tolerances.

This timer-focused benchmarking layer directly complements the session ML work in Section 14: if the time base is stable, then downstream models that reason about pacing, overload, or documentation fraction can rely on accurate temporal features.

15.1.3 Resource Usage for Flows, Timer, and IDE

The flows, timer, and IDE modules exercise the browser in different ways: flows involve dynamic form state and conditional rendering; the timer requires high-frequency updates to the UI; and the IDE combines a Monaco-based editor with background AI activity and a terminal log bus. To ensure that the workbench performs acceptably on commodity hardware, the evaluation strategy defines a small number of composite scenarios and measures resource usage during each.

Let $\mathcal{S}_{\mathrm{bench}}$ denote a set of benchmark scenarios such as:

- *Flow-centric session:* open the psychiatry modal, complete a multi-step agitation or capacity flow, and export a narrative via the tools panel;
- *Timer-centric session:* run a 50-minute therapy session with frequent laps and segment changes while intermittently generating AI summaries; and
- *IDE-centric session:* edit multiple TypeScript files in the embedded IDE, use AI-assisted refactoring several times, and run synthetic flows in a preview panel.

Table 69: *Summary of core technical metrics used in the evaluation strategy. Column widths are bounded to avoid overflow in the printed layout.*

Benchmark target	Primary metric(s)	Instrumentation / source
AI routes (provider–model combinations)	Mean and 95th percentile latency; prompt and completion token counts; estimated cost per request and per thousand tokens.	OpenTelemetry spans and histograms via `withSpan`; counters `req_latency_ms`, `tokens_prompt`, `tokens_completion`, and `cost_usd` in `METRICS`.
Timer engine (stopwatch and countdown)	Mean and maximum drift for target durations $t \in \{5, 15, 30, 50\}$ minutes; symmetry of drift across browsers and hardware profiles.	Comparison of timer-reported elapsed time from `timerEngine` with a high-resolution reference clock (`performance.now()`); optional simulation harness driving synthetic ticks.
Flows and tools modules	Page responsiveness and event-handling latency under complex flows and exports; absence of freeze or jank during navigation.	Correlation of span timings for flow steps and export operations with user-visible events; manual inspection via browser Developer Tools for worst-case scenarios.
Embedded IDE (editor + AI)	CPU and memory footprint during extended editing sessions with repeated AI-assisted refactoring; latency of apply-plan executions.	External resource profiling (task manager, performance panel) combined with span-level timing around AI apply-plan routes and terminal log events.

For each scenario $s \in \mathcal{S}_{\text{bench}}$ we measure resource usage trajectories

$$u_s(t) = \big(\text{CPU\%}_s(t),\ \text{Mem}_s(t)\big),$$

where $\text{CPU\%}_s(t)$ is the fraction of one logical core consumed by the browser process and $\text{Mem}_s(t)$ is the memory footprint, sampled at a fixed interval (for example, every 5 seconds) using browser or operating system tooling. From these trajectories we derive summary statistics such as

$$\overline{\text{CPU}}_s = \frac{1}{T_s}\int_0^{T_s} \text{CPU\%}_s(t)\,\mathrm{d}t, \qquad \overline{\text{Mem}}_s = \frac{1}{T_s}\int_0^{T_s} \text{Mem}_s(t)\,\mathrm{d}t, \tag{15.4}$$

where T_s is the scenario duration. In practice these integrals are approximated by discrete sums over the sampled points. The resulting numbers are compared against thresholds chosen to reflect a realistic baseline (for example, a mid-range laptop with four logical cores and 8–16 GiB of RAM).

Although detailed CPU and memory profiling often requires external tools, the evaluation strategy again leverages existing observability hooks where possible. Long-running operations (such as large exports or batch AI calls) are wrapped in spans, making it possible to correlate resource spikes with specific features or workflows. In future work, lightweight browser-side sampling of heap size or event loop lag could be wired into the existing `METRICS` structure to capture coarse-grained resource metrics alongside latency and token counts.

15.1.4 Summary of Measurement Targets

Table 69 summarises the main technical metrics, their measurement sources, and their primary interpretive roles in the evaluation framework.

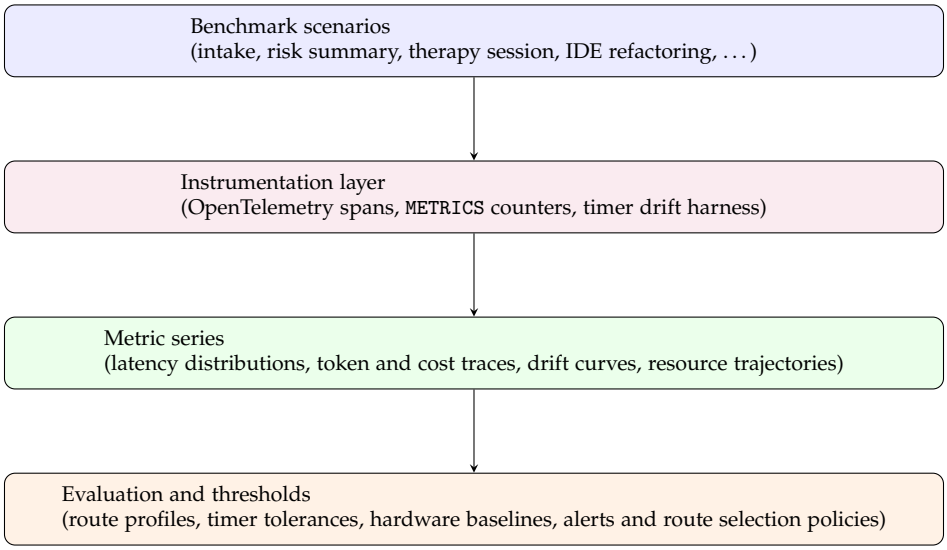


Figure 47: *Conceptual pipeline for technical evaluation. Realistic benchmark scenarios are exercised through the existing instrumentation layer, producing metric series that are summarised into route profiles and acceptance thresholds.*

15.2 Usability Studies

Whereas Section 15.1 focuses on latency, drift, and resource usage, the present subsection addresses the question that ultimately matters to practising clinicians: *does SynapseCore make core psychiatric work easier, faster, and safer without adding cognitive overhead?* To answer this, we adopt a scenario-based usability evaluation strategy that mirrors real clinical use cases and leverages the existing flows framework (Section 7), psychiatry content library, and timer and tools modules.

The overarching design is mixed-methods. Quantitative measures (time-on-task, task success, navigation patterns, System Usability Scale scores) are complemented by rich qualitative feedback obtained via think-aloud protocols and post-session interviews. All data are anchored in concrete interactions with the live application: an intake note is built using the same structured flows, psychometric calculators, and AI routes that the clinician will later use in day-to-day work; risk summaries are generated via the safety and agitation flows; and measurement-based care (MBC) reviews rely on the PHQ-9, GAD-7 and other scale calculators already described in earlier sections.

15.2.1 Study Design and Participant Groups

The usability study targets three primary personas already implicit in the SynapseCore design:

- *Consultant psychiatrist:* senior clinicians who conduct high–acuity assessments and supervise trainees.
- *Early-career clinician:* trainees, residents, fellows, or non-physician mental health professionals who are still building their workflow habits and documentation strategies.
- *Clinical informatics / power users:* psychiatrists or allied professionals with a strong interest in digital tools, who are also likely to interact with the IDE and configuration aspects of SynapseCore.

In a typical evaluation, we recruit $N \approx 12$–18 clinicians, balanced across these personas where possible. Each participant undergoes a brief orientation (10–15 minutes) that mirrors

how the system would be introduced in practice: we demonstrate the psychiatry home view, the flows rail and right dock, the timer modal, and the tools panel for exports. We explicitly avoid extensive training on hidden shortcuts or rare features so that the study reflects the actual discoverability and cognitive ergonomics of the interface.

The core of the protocol is a within-subjects design: each participant completes the same set of scenarios $\mathcal{S}$ (intake note, risk summary, MBC review) using the current version of SynapseCore. Scenario order is randomised for each participant in order to minimise order effects. Within each scenario, participants are encouraged to "work as they normally would" rather than rushing for speed; they are told that the quality and completeness of the clinical artefact (note, summary, or review) matters as much as time and clicks.

During each session, three data streams are collected:

1. *Interaction telemetry:* OpenTelemetry spans and metrics capture navigation between layouts (Section 10), flow step transitions (Section 7), AI route invocations (Section 4), and timer events (Section 14).
2. *Screen and audio capture:* for a subset of sessions, think-aloud protocols are recorded to enable subsequent qualitative analysis of hesitation points, misunderstandings, and emergent strategies.
3. *Self-report instruments:* after completing the scenarios, participants fill out usability questionnaires (including the System Usability Scale) and a short custom survey on trust in AI-assisted documentation and perceived cognitive load.

This design allows formal statistical analysis of quantitative metrics while also supporting deep qualitative insight into why a particular scenario felt "smooth" or "clunky." Because all metrics are grounded in the same telemetry substrate used for routine observability, the evaluation framework can be reused to monitor usability after upgrades or configuration changes.

15.2.2 Scenario-Based Task Definitions

We define a finite set of benchmark scenarios $\mathcal{S} = \{s_{\text{intake}}, s_{\text{risk}}, s_{\text{mbc}}\}$, each corresponding to a clinically meaningful workflow that SynapseCore explicitly supports.

Formally, each scenario $s \in \mathcal{S}$ is represented as a tuple

$$s = \big(G_s, \mathcal{O}_s, \Theta_s\big),$$

where:

- G_s is a directed acyclic graph of *micro-interactions* (opening a specific flow shell, advancing a step, triggering a scale calculator, starting or stopping the timer, invoking an AI route, saving or exporting a note).
- $\mathcal{O}_s$ is the target set of observable outputs (for example, a completed intake narrative attached to the encounter, a structured risk summary in the flows run registry, or an updated MBC dashboard with new scores and risk grades).
- Θ_s is a set of evaluation constraints and expectations (for example, completing risk documentation without omitting critical safety questions, or reviewing psychometric trends spanning at least three visits).

In practice, G_s is not enforced as a strict script; it serves as a reference path that defines what the system considers an "ideal" or canonical use of its own affordances. Participants are free to deviate from this path, and such deviations are precisely the phenomena we wish to observe: do clinicians attempt to document risk in free-text notes instead of using the safety flow? Do they bypass the timer, or do they anchor their work in segments and laps?

Table 70: *Scenario-based tasks used in usability studies. Each scenario exercises a different combination of flows, timer, and tools, and produces a concrete clinical artefact that can be assessed for completeness and clarity. Column widths are chosen to avoid page overflow.*

Scenario	Primary SynapseCore components	Target clinical artefact and focus
Intake note creation	Psychiatry home view; structured flows (e.g. capacity / consent if relevant to presentation); psychometric calculators (PHQ-9, GAD-7, others as appropriate); timer for segmenting history, examination, and plan; tools panel for export.	First-visit or transfer-of-care intake note capturing chief complaint, history, mental status examination, diagnosis or problem list, and initial plan, with explicit linkage to any scales or flows used.
Risk summary	Safety and agitation flows; timer laps to delimit the risk assessment segment; right-dock evidence cards on suicide risk, agitation, and observation levels; AI-assisted summary via Consulton or psychiatry AI panel.	Structured risk formulation (acute and chronic risk, protective factors, agreed safety plan, observation level) that can be reused across notes and handover communications.
MBC review	MBC dashboard and scale calculators; flows for follow-up visits; timeline of scores and risk grades; tools panel for generating MBC-focused patient letters or internal summary notes.	Brief but comprehensive MBC review note that contextualises recent changes in scores and risk grades, links them to treatment adjustments, and highlights any concerning trends requiring closer monitoring.

Table 70 summarises the three primary scenarios and links them to their main UI components and expected outputs.

For each scenario, the study moderator provides a short written vignette (e.g. a distressed new patient with mixed anxiety and depression, an acutely suicidal inpatient, or a stable outpatient with fluctuating scores over the past three months). Participants are asked to imagine that SynapseCore is already integrated into their clinical environment and to work as if they were documenting in the live EHR, while the system silently records the relevant telemetry for subsequent analysis.

15.2.3 Quantitative Usability Metrics and Analysis

To quantify usability, we derive a family of metrics from the interaction telemetry recorded during each scenario. Let u index participants ($u = 1, \ldots, U$) and let $s \in \mathcal{S}$ be a scenario. We define:

- $T_{u,s}$: total time-on-task for scenario s (in seconds), measured from the first interaction in the psychiatry feature to the final export or explicit declaration of completion.
- $C_{u,s}$: number of clinically-relevant content units created, such as completed flows, scale scores, or narrative segments.
- $E_{u,s}$: count of *navigation errors*, where the user moves to an unrelated layout (e.g. leaving the psychiatry feature or opening the IDE) and then returns without a concrete benefit for the task at hand.

- $H_{u,s}$: number of *hesitation episodes*, operationally defined as pauses exceeding a threshold (e.g. 5 seconds) immediately before or after critical steps in G_s, as observed either in screen recordings or via timestamped interaction gaps.

At the scenario level, we summarise these quantities using means and proportions. For example, the mean success rate for scenario s is

$$\hat{p}_s^{\text{succ}} = \frac{1}{U} \sum_{u=1}^{U} \mathbf{1}\{\text{all required outputs in } \mathcal{O}_s \text{ are present}\},$$

and the mean time-on-task is

$$\hat{\mu}_s^{(T)} = \frac{1}{U} \sum_{u=1}^{U} T_{u,s}.$$

In addition, we define an *efficiency index* for each user and scenario by normalising time-on-task to an expert benchmark T_s^{expert}, obtained from one or more pilot users who are already familiar with SynapseCore:

$$\text{Eff}_{u,s} = \frac{T_s^{\text{expert}}}{T_{u,s}},$$

where values $\text{Eff}_{u,s} \approx 1$ indicate expert-level speed, $\text{Eff}_{u,s} < 1$ indicates slower-than-expert performance, and $\text{Eff}_{u,s} > 1$ would indicate unusually rapid completion, which may or may not be desirable depending on the quality of the resulting artefact.

A composite usability score for each (u, s) can then be defined as

$$U_{u,s} = \alpha_T \, \tilde{T}_{u,s} + \alpha_E \, \tilde{E}_{u,s} + \alpha_H \, \tilde{H}_{u,s} + \alpha_C \, \tilde{C}_{u,s},$$

where the tildes denote normalised versions of the raw metrics (e.g. z-scores or min–max rescaling within scenario) and the coefficients $\alpha_T, \alpha_E, \alpha_H, \alpha_C$ reflect the relative weight placed on speed, error avoidance, hesitation, and output density. For descriptive analyses, these composite scores augment, rather than replace, more interpretable single metrics such as median time-on-task or median number of navigation errors.

Because SynapseCore already emits fine-grained spans for key events (AI calls, timer state changes, flow step transitions, exports), the implementation burden for this metric layer is low: each event type is associated with a small number of labels (scenario identifier, participant pseudonym, task phase), and standard scripts aggregate the resulting logs into the summary statistics outlined above.

15.2.4 Standardised Usability Instruments

In addition to behavioural metrics, each participant completes one or more standardised usability instruments immediately after finishing the scenario set. The core instrument is the System Usability Scale (SUS), which yields a score $\text{SUS}_u \in [0, 100]$ for each participant. Let $x_{u,i}$ denote the response of user u to item $i \in \{1, \dots, 10\}$ on a 5-point Likert scale. The standard SUS scoring scheme can be summarised as

$$\text{SUS}_u = 2.5 \left(\sum_{i \in \mathcal{O}} (x_{u,i} - 1) + \sum_{i \in \mathcal{E}} (5 - x_{u,i}) \right),$$

where $\mathcal{O}$ and $\mathcal{E}$ index odd and even items, respectively. Scenario-level SUS summaries ($\overline{\text{SUS}}$ and confidence intervals) provide a global view of perceived usability.

Table 71: *Mapping between latent usability constructs and observable measures in the evaluation framework. Behavioural and self-report measures are designed to be complementary rather than redundant.*

Construct	Behavioural indicators	Self-report indicators
Efficiency and flow	Time-on-task $T_{u,s}$; efficiency index $\mathrm{Eff}_{u,s}$; number of unnecessary layout switches.	SUS items on ease of use and integration; custom items on "getting into flow" with the interface.
Learnability and discoverability	Reduction in $T_{u,s}$ and $E_{u,s}$ between early and late scenarios (if repeated measures are available).	SUS items on learning to use the system quickly; open-text comments on what was hard to find or understand.
Cognitive load	Hesitation episodes $H_{u,s}$; long pauses before critical decisions; frequent revisiting of the same flow step without progress.	NASA-TLX mental demand and effort subscales; custom items on feeling overloaded by information or options.
Trust and perceived safety	Frequency with which AI outputs are edited extensively, discarded, or accepted; use of flows versus free-text for high-stakes tasks.	Custom Likert items on trust in AI summaries and flows, and on perceived support for safe practice.

Where study logistics allow, we also collect a short workload instrument such as the NASA-TLX, focusing on mental demand and effort, and a small set of custom items specific to digital psychiatry, for example:

- perceived support for measurement-based care (e.g. "the system makes it easier to use rating scales systematically");
- trust in AI-generated text (e.g. "I would feel comfortable pasting the AI summary into my note after minor edits"); and
- perceived impact on therapeutic presence (e.g. "the interface does not distract me from the patient").

Table 71 summarises how behavioural metrics and self-report instruments map onto latent constructs of interest.

15.2.5 Qualitative Feedback and Iterative Refinement

Quantitative metrics alone cannot capture the nuanced reasons why a particular layout, label, or flow feels natural or awkward. For this reason, a substantial portion of the usability study is devoted to qualitative feedback.

During the scenarios, participants are encouraged to verbalise their thought processes (think-aloud). After completing all scenarios, a structured debriefing interview explores topics such as:

- where they felt "lost" in the interface and how they attempted to recover;
- which features felt like clear "wins" (e.g. flows that reduced mental bookkeeping, timers that provided a sense of pacing);
- whether AI-generated text aligned with their own documentation style, and how much editing was required; and
- whether they perceived any tension between documentation and therapeutic presence.

Interview recordings are transcribed and coded using a pragmatic thematic analysis.

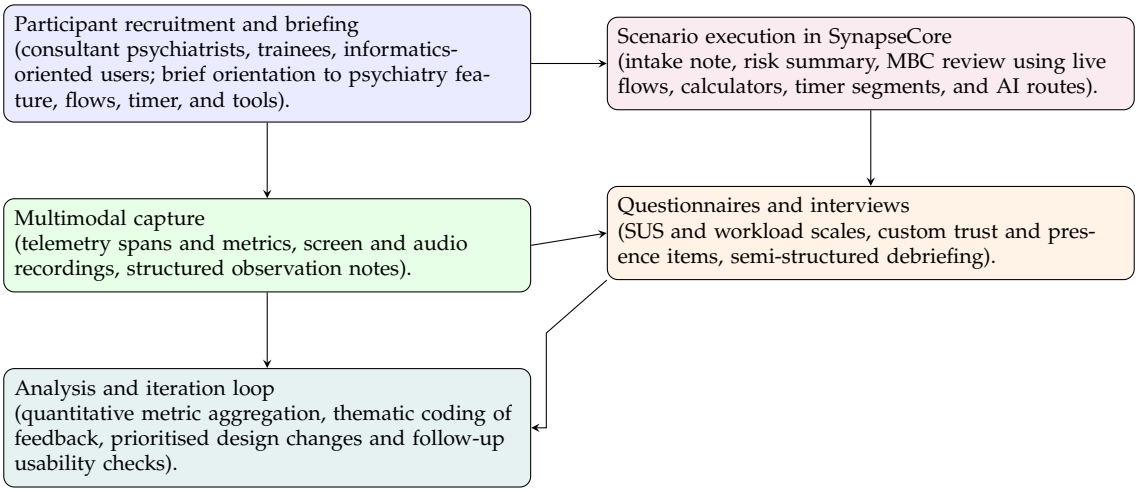


Figure 48: *Usability study pipeline. Clinicians complete realistic scenarios in the live SynapseCore environment while telemetry, recordings, and self-report measures are collected. Quantitative and qualitative analyses feed into an iterative improvement loop, ensuring that flows, timers, and AI-assisted tools support rather than hinder clinical work.*

Codes are anchored in the concrete UI components and flows already documented in earlier sections (for example, `FlowHost` navigation, psychiatry header search, timer-modal segmenting, tools export options). The analysis seeks patterns such as:

- recurrent confusion around particular labels or icons;
- systematic workarounds (e.g. clinicians repeatedly avoiding a given flow in favour of free-text notes); and
- emerging best practices that can be documented and disseminated as usage guidelines.

Findings from this qualitative layer are then fed back into the design and implementation cycle. For example, if participants repeatedly misinterpret a flows boundary disclaimer, the copy can be simplified and tested in a follow-up iteration; if the MBC review scenario reveals that clinicians struggle to visualise trends, additional visual summaries or prompts can be prototyped and re-evaluated.

15.3 Clinical Content Validation

In addition to technical performance and usability (Sections 15.1 and 15.2), SynapseCore must ensure that its clinical content—the cards and sections in the psychiatry library, the scale calculators and their severity mappings, and the flows and AI-generated narratives—is aligned with contemporary psychiatric standards and does not introduce hidden bias or unsafe shortcuts. The present subsection formalises the *clinical content validation* layer, which is implemented as an expert panel workflow running on top of the same versioned content primitives used at runtime.

At a high level, the clinical content validation regime is defined by three principles:

1. *Guideline consistency:* content is grounded in recognised practice frameworks (e.g. measurement-based care conventions, standard diagnostic and risk formulations) and avoids idiosyncratic phrasing that would confuse clinicians.
2. *Transparency of mapping:* the transformation from raw patient input (scale responses, flow answers, structured features) to calculators, severity labels, and narratives can be traced and interrogated by clinicians and auditors.
3. *Versioned accountability:* every clinically-relevant content artefact has an explicit version

identifier and review status, so that changes can be introduced in a controlled and auditable fashion.

The expert panel review process is applied to three primary content categories:

- the *content library* (cards, sections, evidence slices, and their grouping into flows and right-dock content);
- the *scale calculators* (scoring algorithms, severity cut-offs, wording for each severity band, and any derived risk grades); and
- the *flows and AI narratives* (prompts, branching questions, and large-language-model-based textual outputs).

15.3.1 Expert Panel Composition and Workflow

The expert panel is constituted as a small but heterogeneous group $\mathcal{P} = \{p_1, \ldots, p_N\}$ of clinicians and informatics-focused reviewers, typically including:

- at least one consultant psychiatrist with extensive experience in acute care and risk assessment;
- at least one psychiatrist or psychologist focused on measurement-based care and psychometrics; and
- at least one clinical informatics specialist with experience in digital documentation and safety-by-design.

Panel members operate on a shared content registry which mirrors the runtime structure of the psychiatry library. Abstractly, we represent a content artefact as a tuple

$$a = \big(\kappa, v, \theta\big),$$

where

- κ is a unique key identifying the artefact (for example, a section identifier, card identifier, or scale code);
- $v \in \mathbb{N}^3$ is a semantic version triple $v = (v_{\text{major}}, v_{\text{minor}}, v_{\text{patch}})$; and
- θ is the actual content payload (for example, text, configuration parameters, or a scoring function definition).

Each panel member $p \in \mathcal{P}$ assigns ratings $r_{p,a} \in \{0, 1, 2\}$ to artefact a, where:

- $r_{p,a} = 2$: content is clinically appropriate as written;
- $r_{p,a} = 1$: content is acceptable but requires minor edits (e.g. wording refinements, ordering adjustments); and
- $r_{p,a} = 0$: content is not acceptable (e.g. clinically misleading, inconsistent with guidelines, or unsafe).

We define an aggregation operator

$$R(a) = \frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} r_{p,a},$$

and an approval decision function

$$\text{Approve}(a) = \begin{cases} \text{“accept”}, & \text{if } R(a) \geq \tau_{\text{acc}}, \\ \text{“revise”}, & \text{if } \tau_{\text{rev}} \leq R(a) < \tau_{\text{acc}}, \\ \text{“reject”}, & \text{if } R(a) < \tau_{\text{rev}}, \end{cases}$$

with thresholds typically chosen such that $\tau_{\text{acc}} \in (1.5, 2]$ and $\tau_{\text{rev}} \in (0.5, 1.5)$. This structure is deliberately simple; more elaborate inter-rater agreement statistics (e.g. Fleiss' κ) can be computed as needed for research purposes, but the core system design assumes a pragmatic consensus workflow in which reviewers discuss contentious artefacts until a stable version can be adopted.

15.3.2 Validation of Content Library (Cards and Sections)

The content library comprises the atomic and composite information structures that clinicians see in the psychiatry feature: cards in the right dock, sections and subsections in flows, and evidence slices linked to guidelines or reference material. Each such artefact is validated along three axes:

1. *Clinical correctness:* the statements made are factually correct and consistent with current consensus; risk formulations and treatment suggestions are conservative and avoid overconfident language.
2. *Relevance and granularity:* the information shown at a given point in a workflow is neither so sparse as to be unhelpful nor so verbose as to cause cognitive overload.
3. *Tone and framing:* the text avoids stigma, is respectful of patients, and is clearly framed as *supportive* rather than prescriptive guidance.

Formally, we may view the content library as a finite set $\mathcal{A}_{\text{lib}}$ of artefacts

$$\mathcal{A}_{\text{lib}} = \{a_1, \dots, a_M\}.$$

For each artefact a_m and for each axis $d \in \{\text{correct}, \text{granular}, \text{tone}\}$, panel members assign axis-specific ordinal ratings $r_{p,a_m}^{(d)} \in \{0, 1, 2\}$, with aggregated axis scores

$$R^{(d)}(a_m) = \frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} r_{p,a_m}^{(d)}.$$

An artefact is eligible for inclusion in the "production" content set if $R^{(d)}(a_m) \geq \tau_{\text{acc}}^{(d)}$ for all axes d. If any axis falls into the "revise" range, the artefact is sent back to the content editor with structured comments.

Table 72 summarises the key dimensions and provides examples of issues that may be identified during this review process.

Content edits are tracked via the version triple v: patch versions typically capture minor wording changes; minor versions reflect more substantial reorganisation of sections and cards; and major versions indicate structural changes to the content model that may affect flows or AI prompts. The expert panel signs off on each release candidate version, which is then tagged for deployment. Historical versions are retained in the registry to preserve an audit trail.

15.3.3 Validation of Scale Calculators

Scale calculators operationalise measurement-based care within SynapseCore by mapping raw item responses to total scores, severity bands, and, in some cases, downstream risk grades. Let σ index a given scale (e.g. a nine-item depression scale or a seven-item anxiety scale). In code, each scale calculator defines:

- a set of items $I_\sigma = \{1, \dots, n_\sigma\}$;
- a scoring function $f_\sigma : \prod_{i \in I_\sigma} R_i \to \mathbb{N}$, where R_i is the response domain for item i; and

Table 72: *Dimensions for clinical content validation in the psychiatry content library. Each card or section is evaluated along these axes by the expert panel, with explicit ratings and comments. Column widths are bounded to respect the page layout.*

Validation axis	Typical review questions	Examples of potential issues
Clinical correctness	Is the statement consistent with current guidelines and consensus? Does it avoid overpromising or unsupported claims about interventions or prognosis?	Outdated medication recommendations; overly definitive statements about causality; risk formulations that ignore protective factors or underplay severity.
Relevance and granularity	Is the level of detail suitable for the point in the workflow? Does the card focus on information that is actionable in this context?	Long cards with general background text that obscures key points; missing links between evidence and suggested next steps; duplicate information across cards.
Tone, stigma, and framing	Is the language respectful and non-stigmatising? Is it clear that the content is guidance rather than a rigid prescription?	Phrases that pathologise the person rather than the symptoms; categorical language that appears to override clinician judgement; ambiguous phrasing around risk that might be misread as minimising concern.

- a banding function $g_\sigma : \mathbb{N} \to \mathcal{B}_\sigma$, where $\mathcal{B}_\sigma$ is the set of severity labels for that scale (e.g. $\{\text{minimal}, \text{mild}, \dots\}$).

The expert panel validates each of these components:

1. *Scoring correctness:* test cases are constructed to ensure that f_σ faithfully implements the published scoring algorithm (e.g. reversing items where appropriate, handling missingness as per the underlying instrument).
2. *Cut-off thresholds:* the thresholds implicit in g_σ are cross-checked against reference sources, and any deviations (for example, due to local policy) are explicitly documented.
3. *Severity wording and risk grades:* the wording attached to each band $b \in \mathcal{B}_\sigma$ is checked to ensure that it communicates clinical meaning accurately and does not drift into diagnostic labels where the scale is only a screening tool.

We can formalise the banding function as

$$g_\sigma(s) = \begin{cases} b_0, & \text{if } \tau_0^{(\sigma)} \le s < \tau_1^{(\sigma)}, \\ b_1, & \text{if } \tau_1^{(\sigma)} \le s < \tau_2^{(\sigma)}, \\ \vdots & \vdots \\ b_{K_\sigma - 1}, & \text{if } \tau_{K_\sigma - 1}^{(\sigma)} \le s \le \tau_{K_\sigma}^{(\sigma)}, \end{cases}$$

where $\tau_0^{(\sigma)} < \tau_1^{(\sigma)} < \cdots < \tau_{K_\sigma}^{(\sigma)}$ are scale-specific thresholds configured in the calculator. Panel review consists of verifying that these thresholds and the associated band labels $\{b_0, \dots, b_{K_\sigma - 1}\}$ are appropriate.

To validate f_σ and g_σ jointly, we define a set of test vectors $\mathcal{X}_\sigma = \{x^{(1)}, \dots, x^{(L_\sigma)}\}$, where each $x^{(\ell)}$ is a full or partial response pattern $x^{(\ell)} = (x_i^{(\ell)})_{i \in I_\sigma}$ chosen to exercise edge cases (e.g. all items at maximum severity, mixed patterns near cut-offs, and common clinical presentations). For each test vector, we compute the calculator outputs

$$s^{(\ell)} = f_\sigma\big(x^{(\ell)}\big), \qquad b^{(\ell)} = g_\sigma\big(s^{(\ell)}\big),$$

and compare them to panel expectations $s^{*(\ell)}$, $b^{*(\ell)}$. Discrepancies are recorded and must be resolved before the scale calculator is marked as clinically approved.

In addition to numerical correctness, the panel reviews the textual summaries that accompany scales in the UI (for example, short interpretive phrases displayed alongside a severity band or risk grade). These are treated as content artefacts in $\mathcal{A}_{\text{lib}}$ and evaluated using the same axes as in Table 72, with special attention to the risk of overinterpretation (e.g. implying a diagnosis from a screening score alone).

15.3.4 Validation of Flows and AI-Generated Narratives

Flows and AI-generated narratives form the bridge between structured inputs (scale scores, discrete answers, structured clinical decisions) and the final narrative artefacts (notes, consult letters, risk summaries). SynapseCore implements this bridge as a composition of:

- flow definitions (step graphs, prompts, branching logic);
- structured output schemas (e.g. JSON structures with clearly-typed fields for risk, formulation, plan); and
- AI routes and prompts that convert these structured outputs and additional context into natural language text.

Formally, let x denote the structured state produced by a given flow run (including scales, selected options, and free-text fragments), and let c denote contextual information (e.g. encounter type, setting). The narrative generation process can be written as

$$y = \mathcal{N}(x, c),$$

where $\mathcal{N}$ is a composition of deterministic formatting layers and a stochastic language model route $\mathcal{M}_{p,m}$ (as in Section 4). Clinical validation requires that:

1. the prompts and instructions passed to $\mathcal{M}_{p,m}$ faithfully reflect the intended clinical framing (e.g. emphasising that the model is drafting text for clinician review rather than issuing autonomous recommendations);
2. the resulting text y is an accurate, conservative transformation of x and does not hallucinate additional clinical facts or advice; and
3. the narrative respects safety constraints (e.g. avoiding direct instructions to patients, avoiding speculative treatment changes, and clearly flagging uncertainty).

In the expert panel workflow, flows and narratives are validated using curated synthetic cases. For each flow ϕ (e.g. an agitation or safety flow), we define a set of synthetic input states $\mathcal{X}_\phi = \{x^{(1)}, \dots, x^{(L_\phi)}\}$ representing typical and edge-case presentations (for example, high acute risk, high chronic but low acute risk, ambiguous responses, missing information). For each $x^{(\ell)}$ and a fixed context c, the system generates a narrative $y^{(\ell)} = \mathcal{N}(x^{(\ell)}, c)$. Panel members then rate each narrative along three axes:

- *Fidelity to input:* does the narrative accurately reflect the structured inputs (including the presence or absence of protective factors, specific risk features, and agreed safety actions)?
- *Clinical appropriateness:* is the narrative phrased in a way that would be acceptable in an actual medical record?
- *Safety and deference:* does the text clearly present itself as a draft to be reviewed and signed by the clinician, rather than as definitive judgement?

Ratings can again be captured using ordinal scales $q_{p,\ell}^{(d)} \in \{0, 1, 2\}$ for each axis d, with summary scores and decision thresholds defined analogously to those for the content library. Where a narrative is repeatedly judged unsafe or misaligned with clinical expectations,

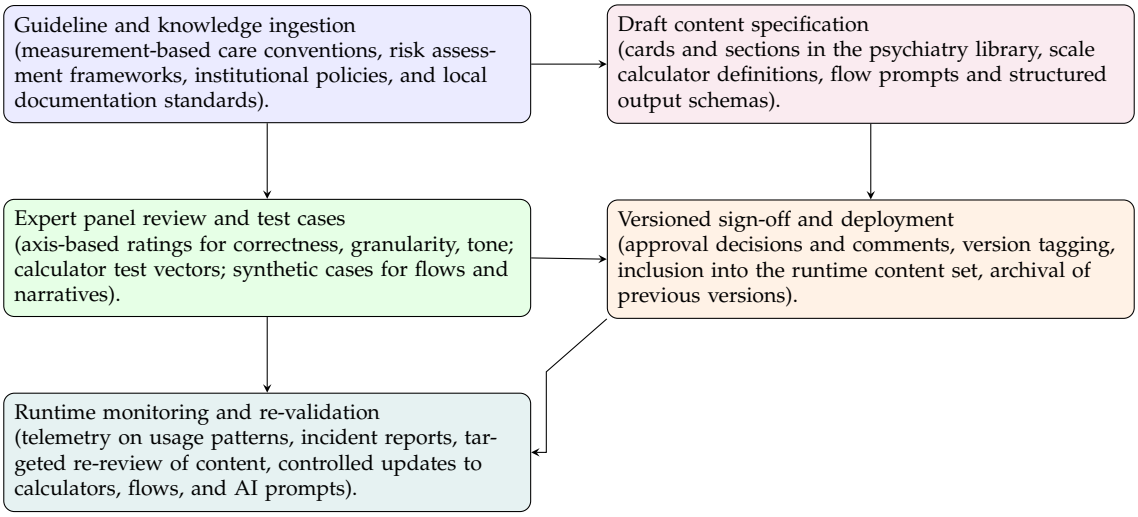


Figure 49: *Clinical content validation pipeline. Source guidelines and institutional policies are translated into draft content artefacts, which are then subjected to expert panel review using structured rating schemes and synthetic test cases. Approved, versioned content is deployed to the runtime system and continuously monitored for emerging issues that may trigger re-validation.*

the panel iterates on the upstream prompt and content configuration (for example, by explicitly instructing the language model to avoid recommending specific medications or by requiring explicit hedging language in high-uncertainty contexts) until acceptable behaviour is observed across the synthetic case set.

15.3.5 Content Validation Pipeline

Figure 49 provides a high-level view of the clinical content validation pipeline. The workflow is deliberately aligned with the runtime content architecture: early stages focus on defining and structuring content artefacts; intermediate stages focus on panel review and iteration; and later stages focus on sign-off and runtime monitoring.

15.4 Safety and Error Analysis

The final layer of the evaluation strategy focuses on safety and error analysis. While Sections 15.1–15.3 address performance, usability, and content fidelity, this subsection treats SynapseCore as a *socio-technical safety system*: AI routes, flows, timers, and tools generate outputs that clinicians may integrate into risk formulations, treatment plans, and medico-legal documentation. Any systematic failure—hallucinated clinical facts, overconfident recommendations, or subtle inconsistencies between structured inputs and generated narratives—poses potential harm.

In SynapseCore, safety is treated as a property of *event streams* rather than isolated calls: every AI invocation, flow execution, export, and error is recorded via the observability infrastructure (Section 12), and these streams are subsequently analysed both manually (expert review of AI outputs) and automatically (targeted analytics over telemetry). This subsection formalises the safety objectives, defines a taxonomy of AI and system errors, and describes how manual review and telemetry-driven analysis jointly monitor and improve the system over time.

15.4.1 Safety Objectives and Threat Model

We begin by specifying a pragmatic threat model. SynapseCore does not execute autonomous treatment decisions; instead, it surfaces structured data, flows, and AI-generated text to a human clinician. Nonetheless, unsafe or misleading outputs can still:

- bias risk formulations (e.g. downplaying suicidal ideation or omitting key risk factors);
- distort diagnostic reasoning (e.g. hallucinating a past diagnosis or mischaracterising symptom trajectories); or
- encourage inappropriate management (e.g. suggesting treatment changes without acknowledging monitoring needs or interaction risks).

To capture these concerns, we define three core safety objectives:

1. *Factual integrity:* AI-assisted text must not introduce unsubstantiated clinical facts that are not present in the structured input or clinician free text.
2. *Conservative guidance:* where the system describes risk, diagnosis, or treatment, it must do so in a cautious manner that defers final judgement to the clinician and avoids prescriptive language.
3. *Robust error handling:* when technical errors occur (provider failures, timeouts, internal exceptions), they must be surfaced clearly to the user and recorded in telemetry for analysis, without silently degrading into misleading partial outputs.

These objectives are enforced at multiple levels: through prompt design in the AI orchestration engine (Section 4), through UI affordances that make it explicit that AI text is a draft to be reviewed, and through the analytical procedures described below.

15.4.2 Event Streams and Error Signals

From an observability perspective, safety analysis operates on a labelled event stream. Let $\mathcal{E}$ denote the set of emitted events from the AI engine and related components. Each event $e \in \mathcal{E}$ is represented as a record

$$e = (t, k, m, \xi),$$

where

- $t \in \mathbb{R}$ is a timestamp,
- $k \in \mathcal{K}$ is a *kind* label (e.g. `ai_request`, `ai_response`, `ai_error`, `flow_run`, `export_error`),
- m is a message payload (e.g. a summary of the error, route identifier, or short description), and
- ξ is a structured metadata object containing fields such as route ID, provider, model, flow identifier, context tags (e.g. `risk`, `mbc`, `intake`), and severity classification if available.

At runtime, these events are emitted via helper functions such as `withSpan` and AI-specific telemetry hooks (for example, a dedicated emitter in `src/observability/aiRouteTelemetry.ts`), which attach consistent attributes to each span and log entry. We can formalise the AI-related subset of the event stream as

$$\mathcal{E}_{\text{ai}} = \{e \in \mathcal{E} : k \in \{\texttt{ai\_request}, \texttt{ai\_response}, \texttt{ai\_error}\}\}.$$

For each AI request, we have a pair of events $(e_{\text{req}}, e_{\text{resp}})$ and, in case of technical failures, possibly an additional `ai_error` event. Telemetry fields in ξ include:

- `route_id`: a provider–model–preset identifier,

- `context_tags`: set of semantic tags derived from the calling component (e.g. `risk`, `intake`, `mbc`, `letter`),
- `status`: success, timeout, provider error, or internal error,
- `token_counts`: prompt and completion token counts, and
- `cost_estimate`: estimated monetary cost (where available).

Safety-specific analyses consume this stream and derive error signals at two levels:

1. *Technical errors:* events where `status` is not success (timeouts, provider 4xx/5xx, internal exceptions).
2. *Content errors:* cases where the output is technically successful but clinically problematic (hallucinations, unsafe or overconfident advice, omissions in high-risk contexts).

Whereas technical errors are detectable directly from telemetry, content errors require manual or semi-automated inspection of AI outputs, as described in the next subsubsection.

15.4.3 Manual Review of AI Outputs

To quantify content-level safety, we construct a labelled dataset of AI interactions through sampling and expert review. Let $\mathcal{C}$ denote the set of candidate AI calls:

$$\mathcal{C} = \{(e_{\text{req}}, e_{\text{resp}}) \in \mathcal{E}_{\text{ai}}^2 : \texttt{status}(e_{\text{resp}}) = \texttt{success}\}.$$

We define a sampling distribution π over $\mathcal{C}$ that overweights high-risk contexts:

$$\begin{aligned}\pi(c) \propto\ & w_{\text{base}} + w_{\text{risk}}\,\mathbf{1}\{\texttt{risk} \in \texttt{context\_tags}(c)\} \\ & + w_{\text{intake}}\,\mathbf{1}\{\texttt{intake} \in \texttt{context\_tags}(c)\} \\ & + w_{\text{mbc}}\,\mathbf{1}\{\texttt{mbc} \in \texttt{context\_tags}(c)\}.\end{aligned}$$

with $w_{\text{risk}} \geq w_{\text{intake}} \geq w_{\text{mbc}} \geq w_{\text{base}} > 0$. Sampling according to π yields a review set $\mathcal{C}_{\text{rev}}$ of size N_{rev}.

For each sampled AI interaction $c \in \mathcal{C}_{\text{rev}}$, expert reviewers (psychiatrists with familiarity in risk and documentation) are given:

- the structured inputs (flow state, scale scores, key free-text notes);
- the prompt and system instructions used for the AI call; and
- the resulting output text as seen in the UI.

Reviewers assign labels on three primary dimensions:

- $h(c) \in \{0, 1\}$: hallucination indicator (1 if the output asserts facts not supported by the input; 0 otherwise);
- $u(c) \in \{0, 1, 2\}$: unsafe advice rating, where 0 means no unsafe content, 1 indicates minor issues that would likely be corrected by a clinician, and 2 indicates potentially harmful or clearly inappropriate suggestions; and
- $m(c) \in \{0, 1, 2\}$: misalignment rating, where higher values indicate that the narrative substantially misrepresents the underlying clinical picture (e.g. omitting major risk factors or exaggerating improvement).

From these labels we can define empirical rates for each AI route r. Let $\mathcal{C}_{\text{rev}}(r) \subseteq \mathcal{C}_{\text{rev}}$ denote the subset of reviewed calls using route r. The hallucination rate and unsafe-advice rate for route r are given by

Table 73: *Safety-oriented taxonomy of errors in SynapseCore and corresponding system responses. Categories are applied both to AI-generated content and to technical failures surfaced via telemetry. Column widths are chosen to respect page margins.*

Error category	Description / examples	Typical system response
Technical fault (non-content)	Provider timeouts, network failures, malformed responses, internal exceptions in orchestration or flow execution.	Clear UI error state; no partial AI text shown without warning; structured `ai_error` event emitted; route-level technical metrics updated; potential automatic retry with safe defaults.
Benign content deviation	Minor stylistic issues or slight over-elaboration that do not change clinical meaning and are easily corrected by the clinician.	No immediate system change; issue may inform future prompt tuning or templates; optional low-severity flag for analysis.
Hallucinated but clinically neutral content	AI text introduces unsupported details that are not clinically consequential (e.g. inferred but non-critical background).	Route-level hallucination index updated; prompts adjusted to emphasise avoiding speculation; may trigger targeted re-review if frequency increases.
Clinically misleading narrative	Misrepresentation of severity, risk, or trajectory (e.g. stating "marked improvement" despite worse scores, or omitting high-risk features).	Marked as medium-severity error; prompts and structured-to-text mapping reviewed; affected route may be restricted from high-risk contexts until fixed and re-validated.
Potentially unsafe advice or instructions	Text that could be interpreted as recommending specific treatment changes, advising patients directly, or minimising acute risk.	High-severity safety incident; route disabled or restricted; prompts tightened to prohibit direct advice; incident logged and may trigger broader content and configuration review.

$$\widehat{\mathrm{Hall}}_r = \frac{1}{|\mathcal{C}_{\mathrm{rev}}(r)|} \sum_{c \in \mathcal{C}_{\mathrm{rev}}(r)} h(c), \tag{15.5}$$

$$\widehat{\mathrm{Unsafe}}_r = \frac{1}{|\mathcal{C}_{\mathrm{rev}}(r)|} \sum_{c \in \mathcal{C}_{\mathrm{rev}}(r)} \mathbf{1}\{u(c) = 2\}. \tag{15.6}$$

Analogously, a misalignment index can be computed from $m(c)$. These indices become part of the route-level profile alongside technical metrics from Section 15.1. For a route to be eligible for clinical use in high-risk contexts, we might require, for instance, that $\widehat{\mathrm{Unsafe}}_r \approx 0$ in the review set and set policy thresholds such as:

$$\widehat{\mathrm{Hall}}_r \le \varepsilon_{\mathrm{hall}}, \qquad \widehat{\mathrm{Unsafe}}_r = 0,$$

for a specified small $\varepsilon_{\mathrm{hall}} > 0$. Routes exceeding these thresholds are either removed from clinical contexts, restricted to low-stakes documentation tasks, or subjected to prompt and configuration revisions followed by re-review.

15.4.4 Taxonomy of Errors and Response Matrix

Not all errors have equal weight. SynapseCore therefore organises observed issues into a safety-oriented taxonomy, with a corresponding system response for each class. Table 73 summarises the main categories.

In practice, high-severity cases are rare but treated as sentinel events: they trigger a small "safety council" review, which may include additional internal or external experts beyond the

core development team. The telemetry infrastructure ensures that each such event is accompanied by full context (route, prompts, structured inputs, and UI state) to enable thorough reconstruction.

15.4.5 Telemetry-Based Monitoring and Near-Miss Detection

While manual review produces detailed labels on a subset of AI calls, the broader error landscape is monitored using aggregated telemetry and clinician-facing reporting affordances. SynapseCore includes a lightweight "Report unsafe or misleading AI output" mechanism within AI-assisted panels; when activated by a clinician, this emits a high-priority event

$$e_{\text{report}} = (t, k = \texttt{ai\_report}, m, \xi_{\text{report}}),$$

where m captures a short free-text rationale and ξ_{report} includes route ID, context tags, and local identifiers for the underlying encounter (internally pseudonymised in exports).

Let N_r denote the total number of AI calls for route r in a given time window, and let R_r denote the number of clinician reports associated with that route. A simple near-miss rate can then be defined as

$$\widehat{\text{NM}}_r = \frac{R_r}{N_r}.$$

This signal is imperfect: clinicians may under-report subtle issues or over-report in early phases when trust is still forming. However, when combined with manual review indices $\widehat{\text{Hall}}_r$, $\widehat{\text{Unsafe}}_r$ and with technical error rates, it provides a practical way to prioritise routes and contexts for deeper investigation.

In addition, telemetry from flows and tools is used to detect patterns such as:

- unusually high rates of AI cancellations or deletions after generation in a given context (suggesting unsatisfactory outputs);
- repeated re-generation loops (clinicians pressing "regenerate" multiple times on the same prompt); and
- disproportionate avoidance of AI features in specific flows (e.g. clinicians consistently opting out of AI assistance for risk narratives).

These patterns are treated as soft safety signals: they may indicate that AI outputs are perceived as unhelpful or misaligned with clinical expectations, even when they do not cross the threshold into clearly unsafe content.

15.4.6 End-to-End Safety Pipeline

Figure 50 summarises the end-to-end safety and error analysis pipeline in SynapseCore. Conceptually, it mirrors the technical evaluation and clinical validation pipelines, but with an explicit focus on the identification and mitigation of both technical and content-level risks.

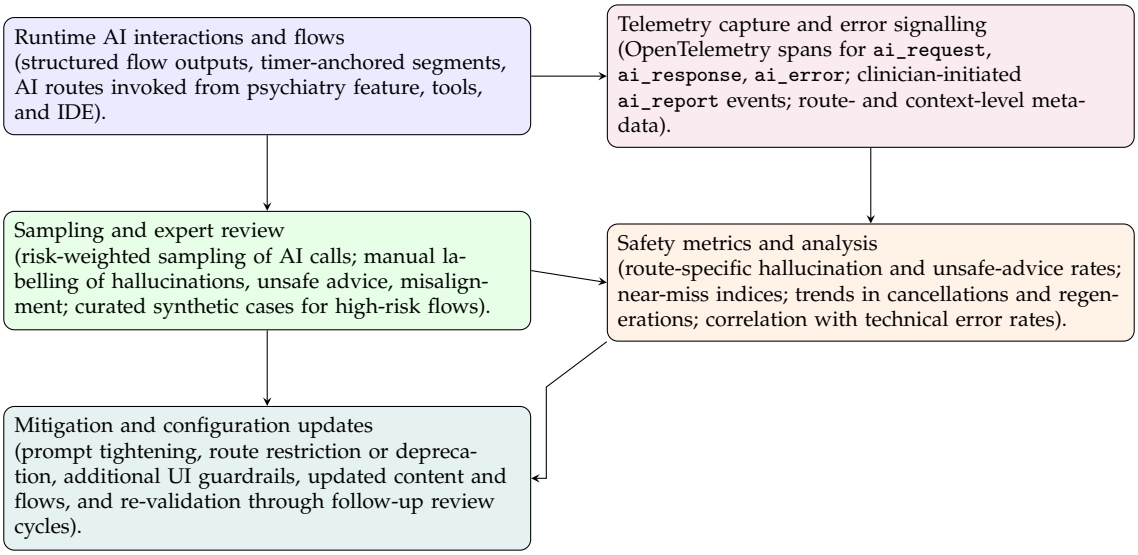


Figure 50: *Two-column compact schematic of the safety and error analysis pipeline. Runtime AI flows and telemetry form parallel streams that converge through expert review and metric computation, informing mitigation, guardrails, and follow-up validation.*

16 Ethical, Legal, and Regulatory Considerations

SynapseCore is intentionally positioned as a programmable digital psychiatry workbench rather than a fully autonomous clinical decision system. This distinction is not only technical (Section 4, Section 6, Section 7, Section 12), but also normative: the platform is designed so that licensed clinicians remain epistemic and moral agents, while the software provides structured views, measurements, and AI co-pilot functionality that can be inspected, overridden, or discarded. In line with contemporary guidance on artificial intelligence in healthcare and mental health (Olawade et al., 2024; World Health Organization, 2021, 2023), the overarching ethical design goals are to preserve human oversight, support patient autonomy, minimise harm, promote fairness, and maintain accountability across the full sociotechnical system rather than relying on any single algorithmic component.

At a high level, the workbench mediates between three overlapping domains: (i) clinical practice in psychiatry and mental health services, (ii) software and data protection law (to be elaborated in subsequent subsections on data privacy, security, and regulatory pathways), and (iii) institutional policies and professional ethics codes. For each domain, the architecture purposely encodes *friction* at decision points where uncritical automation would be risky. For example, flows in Section 7 require explicit clinician confirmation before committing documentation; MBC scores in Section 6 are presented alongside raw item-level responses; and AI-generated narratives are rendered as draft text rather than self-executing orders or diagnoses. These structural choices are intended to align with widely endorsed principles of human-in-the-loop AI, transparency, and contestability in clinical settings.

Formally, let $\mathcal{D}$ denote the space of clinical data objects encapsulated by the workbench (measurement-based care vectors, flow states, free-text notes, timer events), and let $\mathcal{S}$ denote the space of AI suggestions and generated artefacts (summaries, letters, risk narratives, psychoeducational text). The AI orchestration engine (Section 4) instantiates a stochastic operator

$$\mathcal{O} : \mathcal{D} \times \mathcal{C} \to \mathcal{P}(\mathcal{S}),$$

where $\mathcal{C}$ collects configuration and context (selected provider, model, sampling parameters, prompt templates, and additional context bundles) and $\mathcal{P}(\mathcal{S})$ denotes probability measures

over outputs in $\mathcal{S}$. The clinician then applies a selection functional

$$\Phi_{\text{clin}} : \mathcal{D} \times \mathcal{S} \to \mathcal{A},$$

mapping the incoming data and candidate AI outputs into concrete clinical actions $\mathcal{A}$ (such as diagnostic formulations, treatment plans, documentation entries, or explicit decisions not to act). The *intended* role of SynapseCore is to constrain $\mathcal{O}$ and the affordances around Φ_{clin} so that every AI-mediated action remains (i) reviewable in context, (ii) attributable to a licensed professional, and (iii) embedded within local clinical governance mechanisms.

The following subsections make this informal commitment explicit. Section 16.1 articulates the intended use and scope of SynapseCore as an educational, research, and prototyping workbench under clinician supervision. Subsequent subsections (to be developed) will address data protection, security, and regulatory classification in more detail, connecting the technical design in Sections 4–14 with emerging guidance on AI for health and the specific vulnerabilities of psychiatric populations.

16.1 Intended Use and Scope

From a regulatory and ethical perspective, *intended use* is not a mere declarative statement placed in marketing materials; it is a formal specification that constrains how a sociotechnical system may be safely deployed. This notion of intended use is central to modern regulation of clinical decision support and software-as-a-medical-device (U.S. Food and Drug Administration, 2022a). In the current iteration, SynapseCore is explicitly designed and documented as:

- an *educational and training environment* for psychiatry and mental health, in which trainees and clinicians can explore structured flows, measurement-based care (MBC), and AI-assisted documentation using synthetic or de-identified cases;
- a *research and prototyping workbench* for clinical informatics, allowing teams to design and evaluate new flows, prompts, scoring strategies, and AI orchestration routes without direct, real-time impact on patient care; and
- a *supervised documentation and reasoning aid* that may be used within routine clinical practice to draft notes, construct summaries, and organise MBC information under the continuous oversight of a licensed clinician who remains responsible for every diagnostic and treatment decision.

Crucially, SynapseCore is *not* intended to function as an autonomous diagnostic engine, triage gatekeeper, or prescribing system. The platform is not designed to directly ingest raw sensor streams or patient-facing chat logs and then emit unsupervised clinical decisions. Instead, the architecture assumes that (i) clinicians control which data are imported into the workbench, (ii) all AI outputs are presented as draft artefacts within a broader clinical reasoning process, and (iii) final decisions, orders, and communications are made and signed off by human professionals in accordance with local law and institutional policy.

User roles and supervision

In keeping with professional codes of ethics in psychiatry and clinical psychology, we assume a multi-layered user model:

- **Attending psychiatrists and licensed independent practitioners** (e.g. clinical psychologists, advanced psychiatric nurse practitioners) are the primary responsible users. They may use SynapseCore to structure interviews, interpret MBC scores, draft formulations,

and generate patient-facing psychoeducation, but they remain accountable for the correctness, appropriateness, and communication of all content.

- **Trainees and supervised clinicians** (residents, fellows, psychology trainees) may use the workbench for learning, rehearsal, and supervised documentation. In such contexts, all substantive clinical artefacts generated with SynapseCore (notes, letters, treatment plans) must be reviewed and formally approved by a supervising clinician before entering the medical record or influencing care.
- **Clinical informatics, quality-improvement, and research teams** may deploy SynapseCore on synthetic, anonymised, or appropriately de-identified datasets for exploratory data analysis, workflow design, and prototyping of AI-enhanced tools, subject to institutional review and governance processes.
- **Patients and carers** are *not* direct users of the core workbench in its current design. While patient-facing outputs (for example psychoeducational handouts or appointment summaries) may be derived from SynapseCore, these artefacts are intermediated by clinicians who validate and customise the content.

This user model supports a layered safety strategy: the highest-risk actions (diagnoses, prescriptions, involuntary treatment decisions) are never triggered directly by the software; instead they arise from clinician judgement informed, but not determined, by the tools in Sections 6–7 and the AI panel in Section 4. In formal terms, let $\mathcal{U}$ denote the set of *user roles* and $\mathcal{T}$ the set of *task types* exposed by the workbench (for example scaling questionnaires, structuring HPI, generating summaries, drafting letters, configuring flows). SynapseCore is configured so that the composite map

$$\Psi : \mathcal{U} \times \mathcal{T} \to \mathcal{A}$$

factors through a supervision relation $R \subseteq \mathcal{U} \times \mathcal{U}$, such that any high-risk action $a \in \mathcal{A}_{\text{high}} \subset \mathcal{A}$ can only be realised when there exists a pair $(u_{\text{jun}}, u_{\text{sen}}) \in R$ with u_{sen} a fully licensed clinician who has reviewed and accepted the relevant artefacts.

Permitted task classes

Given this user model, it is useful to make explicit which task classes are within scope for SynapseCore and which are deliberately excluded in the current design.

Within scope:

- *Documentation support and summarisation*, including: drafting of assessment and progress notes, discharge summaries, referral letters, and psychoeducational material using AI models configured through the orchestration engine in Section 4.
- *Measurement-based care support*, including: scoring, banding, and trend visualisation for scales implemented in Section 6, automated insertion of score summaries into notes, and generation of patient-facing explanations of score meanings.
- *Structured risk and outcome documentation*, including: support for the construction of structured narratives based on flows for agitation, capacity, catatonia, observation, and safety (Section 7), together with timer- and event-based annotations from Section 14.
- *Educational simulation and rehearsal*, for example: running through synthetic or anonymised cases to practice assessment flows, MSE documentation, and treatment-plan construction, optionally with AI-generated variants to expose trainees to diverse clinical scenarios.
- *Clinical informatics prototyping*, including: iterative refinement of prompts, flows, scoring rules, and export templates in collaboration between clinicians and technical staff, as described in Sections 4–12.

Table 74: *Intended use matrix for SynapseCore, distinguishing primary user groups, task categories, and indicative regulatory characterisation. Column widths are bounded to avoid overflow in the two-column layout.*

User group	Typical tasks	Clinical context	Indicative regulatory stance
Attending psychiatrists / licensed clinicians	MBC review and note drafting; structuring risk narratives; generating discharge summaries and psychoeducation	Direct patient care with clinician in the loop	Clinician-facing decision and documentation support; designed to fall within non-autonomous, oversight-preserving CDS where clinicians can independently review the basis of AI outputs.
Trainees under supervision	Drafting assessments and plans; rehearsing flows on synthetic or de-identified cases; experimenting with prompts	Education and supervised clinical training	Educational tool; outputs used only after review by a licensed supervisor; not intended as standalone medical advice.
Informatics / research teams	Designing new flows and prompts; prototyping scoring and export logic; running what-if analyses on curated datasets	Quality improvement, service design, and research	Prototyping and research environment; deployment into production clinical workflows requires additional validation, governance, and regulatory review.

Out of scope for the current implementation:

- *Fully autonomous diagnosis or treatment selection* without clinician verification.
- *Standalone patient-facing chatbots* that present as clinicians or imply that they are providing individualised medical advice without human oversight.
- *Automated triage, risk prediction, or resource allocation tools* whose outputs are used as the sole or primary basis for decisions about admission, discharge, involuntary treatment, or access to scarce services.
- *Real-time monitoring of wearable or sensor data* with automatic actuation (for example triggering emergency services) without intermediate clinical review.

Table 74 summarises these categories in terms of primary user groups, example tasks, and indicative regulatory characterisation. The table is intentionally conservative: many tasks that could, in principle, be automated are treated as clinical decisions-in-context requiring explicit human oversight rather than algorithmic control.

Conceptual layers of intended use

Figure 51 provides a schematic view of the intended use layers embedded in the current design. The top layer emphasises that SynapseCore operates as a *workbench* rather than a black-box decision-maker. The middle layers distinguish three main contexts of use (education, research/prototyping, supervised clinical documentation), and the bottom layer explicitly lists excluded uses (such as unsupervised patient-facing agents or fully autonomous triage).

Alignment with emerging ethical and regulatory guidance

Finally, it is important to situate this intended use framing within emerging ethical and regulatory guidance on AI in health and mental health care. Contemporary analyses of AI for mental health emphasise both the potential benefits of improved access, personalisation, and measurement, and the heightened risks of harm, stigma, and loss of autonomy in already vulnerable populations (Ali et al., 2025; Cruz-Gonzalez et al., 2025; Fiske et al., 2019; Olawade et al., 2024). In response, major guidance documents and reviews converge on several re-

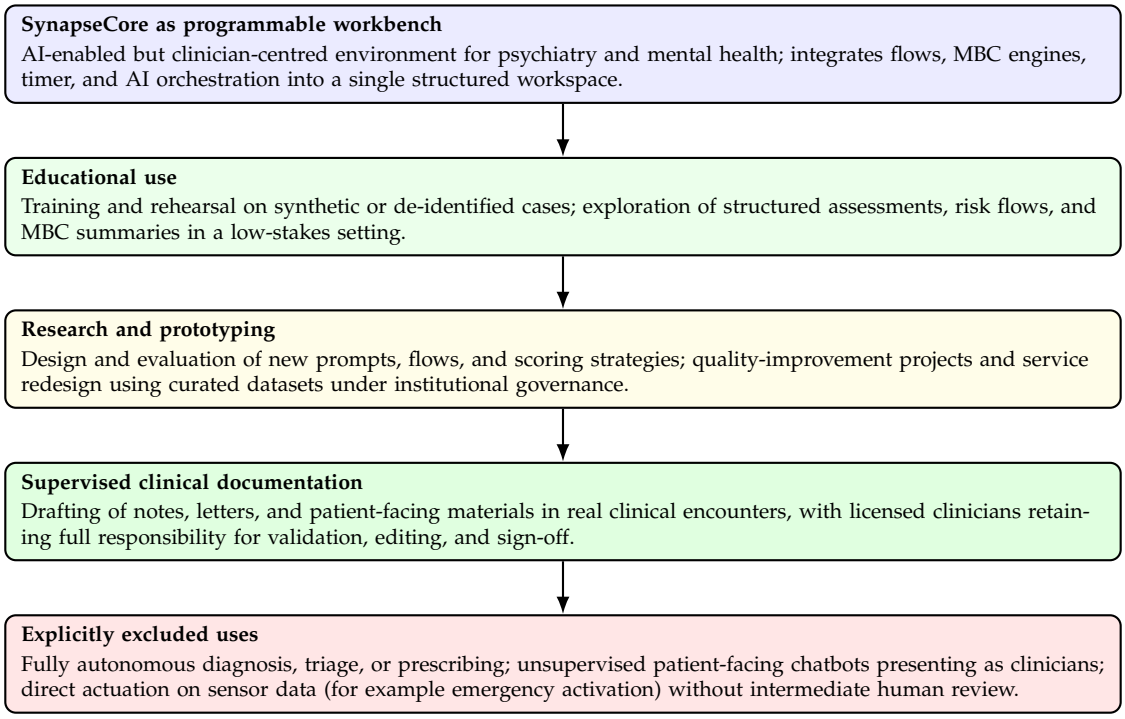


Figure 51: *Conceptual layers of intended use in the current SynapseCore design. The system is framed as a programmable workbench whose primary roles are educational, research/prototyping, and supervised documentation support, with high-risk autonomous functions explicitly excluded.*

quirements: (i) maintain human oversight and clinical accountability, (ii) make the role of AI explicit and understandable to clinicians and patients, (iii) avoid opaque high-stakes automation, and (iv) subject high-risk use-cases to rigorous validation and governance (Elgin & Elgin, 2024; Jones et al., 2023; World Health Organization, 2021, 2023, 2025).

By constraining SynapseCore to educational, prototyping, and supervised documentation roles, and by engineering structural points where clinicians must actively review and confirm AI-mediated suggestions, the present design aims to respect these requirements. Subsequent subsections will build on this foundation by analysing data protection and security and by mapping concrete deployment scenarios to jurisdiction-specific regulatory categories, including potential classifications as clinician-facing clinical decision support tools versus software as a medical device, depending on how local institutions elect to deploy and govern the workbench.

16.2 Data Privacy and Security

This subsection specifies the privacy and security model implemented in the current browser-based React/TypeScript SynapseCore build, with particular emphasis on: (i) client-side storage boundaries and data classes, (ii) configurable de-identification (*de-ID*) flags and transforms, (iii) privacy-aware local logging and observability, and (iv) deployment recommendations for self-hosted instances in which no raw protected health information (PHI) leaves the local environment without explicit, informed consent from the responsible clinician or institution. The goal is to align the engineering primitives of SynapseCore with widely accepted regulatory and professional standards for handling psychiatric data, including the HIPAA Privacy Rule and de-identification guidance in the United States, 42 C.F.R. Part 2 protections for substance use disorder (SUD) treatment records, and the European Union General Data Protection Regulation (GDPR) concepts of anonymisation and pseudonymisation.(European

Parliament and Council of the European Union, 2016a, 2016b; U.S. Department of Health and Human Services, 2024; U.S. Department of Health and Human Services, Office for Civil Rights, 2012)

16.2.1 Clinical Privacy Risks and Threat Model

From a psychiatric perspective, the privacy stakes are unusually high. Encounter notes may contain explicit descriptions of suicidal or homicidal ideation, trauma histories, sexual behaviour, forensic details, substance use, and family dynamics. Leakage of such information can cause direct harm (stigma, discrimination, legal consequences) and indirect harm (loss of trust in care, avoidance of treatment). Digital mental health tools have repeatedly been shown to share sensitive data with third parties without adequate transparency, underscoring the need for cautious, clinician-controlled designs.(Huckvale et al., 2019; Torous et al., 2018)

In SynapseCore, we therefore adopt a deliberately conservative threat model for the reference implementation:

- The browser environment is treated as the primary trusted runtime for interactive use.
- External AI providers (OpenAI, Anthropic, Google, local Ollama, etc.) are considered *conditionally trusted* and must be explicitly configured; the default assumption is that no raw PHI is sent unless the clinician has enabled this and has the appropriate legal and organisational basis.
- Network egress beyond the clinician's machine or local network (e.g. via cloud API calls) is treated as a boundary requiring explicit consent, configuration, and documentation.
- Logging and telemetry subsystems are designed so that they *never require* the collection of free-text clinical content for their basic function; only aggregates, counters, and route metadata are needed for most performance and safety monitoring.

This threat model is operationalised via the client-only storage design, de-identification flags, and a disciplined separation of data classes, as detailed in the following subsections.

16.2.2 Data Classes and Client-Side Storage Boundaries

We define three primary data classes relevant to privacy:

1. *Raw clinical content* (D_{raw}): free-text notes, narrative summaries, psychometric item-level answers, and structured flow data that directly describe an identifiable patient or session.
2. *Derived de-identified content* (D_{safe}): transformed versions of D_{raw} where direct identifiers and quasi-identifiers have been removed, generalised, or pseudonymised in such a way that the risk of re-identification is substantially reduced under the relevant legal standard (e.g. HIPAA Safe Harbor or GDPR anonymisation).(European Parliament and Council of the European Union, 2016b; U.S. Department of Health and Human Services, Office for Civil Rights, 2012)
3. *Operational metadata* (M): timing events, AI provider route information, counters (e.g. token usage, number of calls), user interface preferences, and configuration flags without clinical content.

Formally, we treat the de-identification pipeline as a function

$$T_{\text{deid}} : D_{\text{raw}} \times F \longrightarrow D_{\text{safe}}, \tag{16.1}$$

where F is the set of de-identification flags (e.g. "strip names", "coarsen dates", "remove

Table 75: *Privacy-relevant data classes and client-side storage boundaries in SynapseCore.*

Data class	Typical examples	Primary storage	Persistence policy
Raw clinical content D_{raw}	Live encounter notes; structured flow state (e.g. agitation, capacity); psychometric answers (PHQ-9, GAD-7); draft letters with identifiers	In-memory React state (component tree, context stores)	Ephemeral by default; cleared on page refresh or explicit reset; never written to localStorage automatically.
Derived de-identified content D_{safe}	De-identified AI prompts; synthetic examples; training data for teaching/demo; pseudonymised transcripts	In-memory state; optional local file export at clinician request	Persisted only when clinician explicitly triggers an export or save action; may be stored as local files under institutional policy.
Operational metadata M	Timer laps and durations; AI provider and model choices; token counts; toggles and preferences; anonymised telemetry events	Browser localStorage; in-memory telemetry buffers	Persisted locally for usability and performance; does not contain PHI; can be reset via "Clear data" actions or browser tools.

addresses", "remove organisational identifiers"). The corresponding logging projection

$$L : D_{\text{raw}} \cup D_{\text{safe}} \cup M \longrightarrow M \tag{16.2}$$

is constrained such that for all inputs,

$$L(x) \in M \quad \text{and} \quad x \notin L(x), \tag{16.3}$$

i.e. logs never contain verbatim clinical content; only derived metadata are persisted.

In the reference SynapseCore implementation, these classes map to concrete storage primitives as summarised in Table 75.

Under this model, there is no backend database or cloud service controlled by SynapseCore that stores PHI by default. All state lives either in memory (for D_{raw} and D_{safe}) or in browser-local storage (for M), giving the clinician or institution full control over whether and how data are persisted beyond the session.

16.2.3 De-Identification Flags and Privacy Modes

Clinically meaningful deployment requires that the same codebase can operate under different privacy regimes: from *live clinic* scenarios where PHI is present, to *teaching* and *sandbox* modes where only synthetic or de-identified data should ever be visible. SynapseCore therefore introduces a set of de-identification flags and composite "privacy modes" that influence how data are transformed before being routed to AI providers or export tools.

Let $F = \{f_1, \ldots, f_k\}$ denote the set of atomic de-identification flags. Examples include:

- f_{name}: remove or pseudonymise personal names (patient, relatives, clinicians).
- f_{date}: replace absolute dates with relative descriptors (e.g. "3 weeks ago") or month-year only.
- f_{geo}: coarsen addresses and locations (e.g. city or region instead of street address).
- f_{org}: remove organisational identifiers (hospital names, ward numbers, school names).
- f_{id}: strip explicit identifiers (record numbers, social security numbers, phone numbers, email addresses).

Table 76: *Illustrative privacy modes for different SynapseCore deployment scenarios.*

Mode	Typical use case	Active de-ID flags F_p	Governance constraints G_p
Live clinic (cloud AI off)	Local note-taking, flows, timer; no AI calls	Optional flags (clinician-controlled); transformations applied mainly for exports	All AI routes disabled; no network egress; logging restricted to metadata M.
Live clinic (cloud AI on)	AI-assisted note drafting and psychoeducation with PHI present	Names, IDs, detailed geocodes stripped or pseudonymised; dates coarsened as required	Outbound AI calls permitted only to configured providers; compliance with data processing agreements; logs must not store free text.
Teaching / sandbox	Classroom demonstrations with synthetic or heavily de-identified examples	Full set of de-ID flags (names, dates, geocodes, IDs, organisations) enforced	Network egress permitted for D_{safe} only; no persistent logging of clinical content.
Research prototype	De-identified retrospective data, e.g. for method development	Strong date and location coarsening; removal of rare combinations of attributes	Additional oversight by research ethics board; no linkage keys stored within SynapseCore.

A privacy mode is then a tuple

$$p = (F_p, G_p), \tag{16.4}$$

where $F_p \subseteq F$ is the active set of de-identification flags and G_p specifies additional governance constraints, such as "no network egress" or "allow network egress only for D_{safe}".

At runtime, when a clinician or user attempts to send content to an AI route, SynapseCore applies:

$$D_{\text{out}} = T_{\text{deid}}(D_{\text{raw}}, F_p), \tag{16.5}$$

and verifies that any outbound request satisfies the policy encoded in G_p. For example, a "training / teaching" mode may enforce G_p = "no PHI in prompts and no logging of free text", while an "on-prem clinical" mode may allow richer content but require local-only models.

Table 76 sketches a representative set of modes for different use cases.

In psychiatric contexts where SUD treatment records or other highly sensitive categories are involved, further restrictions may be needed under 42 C.F.R. Part 2 (e.g. strict consent, redisclosure limitations, special accounting of disclosures).(U.S. Department of Health and Human Services, 2024) The mode system is designed so that such jurisdiction- or institution-specific constraints can be expressed as additional governance predicates G_p without modifying the core application logic.

16.2.4 Local Logging, Telemetry, and Auditability

Modern AI systems benefit from telemetry: measuring latency, error rates, token usage, and route selections in order to improve performance and detect failures. At the same time, any logging of psychiatric data must remain tightly scoped. SynapseCore therefore adheres to the following principles:

1. *Log metadata, not notes.* Telemetry events capture which AI route was used, approximate token counts, and timing information, but never the raw text of prompts or responses.

2. *Confine logs to the client.* In the reference build, logs are stored in browser localStorage or in-memory buffers only; there is no default backend telemetry service.
3. *Provide explicit reset and export controls.* Clinicians can clear telemetry and local preferences via UI actions, and (in institutional deployments) export anonymised aggregates for quality improvement or research.

Let E denote the set of telemetry events and C the space of clinical texts. The logging function can be formalised as

$$\ell : C \times P \longrightarrow E, \tag{16.6}$$

where P represents the relevant configuration (AI route, privacy mode, timestamp, etc.). The crucial constraint is that

$$\forall c \in C,\ \forall p \in P: \quad \text{content}(\ell(c,p)) \cap C = \varnothing, \tag{16.7}$$

i.e. the event payload never contains the original clinical text. Instead, it contains summary metrics such as token counts, elapsed time, or categorical error codes.

From a governance viewpoint, this design allows institutions to reason about technical performance and safety signals without creating a secondary dataset of clinical notes that would itself require strong PHI controls and retention policies.

16.2.5 AI Provider Egress Policy and Consent

Because SynapseCore is designed to orchestrate different AI providers, a clear egress policy is required. We distinguish three levels of AI execution environment:

- **Local models** (e.g. on-prem LLMs or small models running on the clinician's machine). These are treated as extensions of the local environment; PHI may be sent under the same institutional safeguards as any other on-prem system.
- **Institutional cloud** (e.g. hospital-managed virtual private cloud with business associate agreements). PHI may be sent if (a) the model provider is under an appropriate data processing agreement, (b) encryption in transit and at rest is enforced, and (c) access controls and logging satisfy local policy.
- **Public cloud APIs** (e.g. consumer accounts for commercial AI APIs). By default, SynapseCore should be configured so that no PHI is sent to such services unless the clinician explicitly enables this in a dedicated configuration screen and confirms that they have the legal and ethical basis to do so.

We can represent the decision to transmit content x to provider p as a predicate

$$\text{send}(x,p) = \begin{cases} \text{true}, & \text{if } x \in D_{\text{safe}} \text{ and } p \in \mathcal{P}_{\text{allowed}}, \\ \text{false}, & \text{otherwise}, \end{cases} \tag{16.8}$$

where $\mathcal{P}_{\text{allowed}}$ is the set of providers configured as acceptable in the current privacy mode. In stricter modes, the condition may further require a recently recorded consent or a documented legal basis. For example, in a GDPR context, sending de-identified but potentially re-identifiable data may require a specific legitimate interest assessment or explicit consent depending on the exact degree of anonymisation and context.(European Parliament and Council of the European Union, 2016a, 2016b)

16.2.6 Vertically Structured Privacy Stack

Figure 52 presents a vertically organised, colour-coded schematic of the SynapseCore privacy stack. The pipeline emphasises that all transformations stay within the client, with

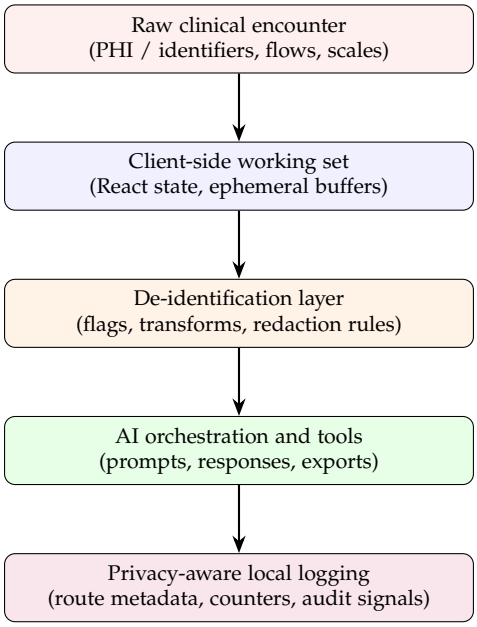


Figure 52: *Vertically structured privacy stack in SynapseCore. Raw encounter data are kept in local memory, transformed by de-identification rules before any AI call or export, and observed only via metadata-level logging for performance and safety.*

de-identification and governance layers interposed between raw clinical content and any AI or logging subsystem.

The narrow horizontal extent of the boxes ($\approx 6.2\,\text{cm}$) ensures that the figure remains well within typical article margins while still providing enough space for legible labels.

16.2.7 Deployment Recommendations for Self-Hosted Instances

The client-only architecture of SynapseCore makes it natural to deploy as a self-hosted web application, either:

- on a clinician's workstation (e.g. via a static file server or container running locally), or
- on an institutional intranet (e.g. hospital virtual private network) with access controlled by existing authentication systems.

For such deployments, we recommend the following baseline controls:

1. **Network architecture.** Host the static assets on an internal server or as a local application bundle. Restrict outbound network access from the workstation or container to explicitly approved AI endpoints and update servers.
2. **Key management.** Store API keys and other secrets in OS- or browser-level secure storage where possible (e.g. encrypted keyrings), rather than in plain-text configuration files. Provide institutional guidance on which keys may be used for PHI-bearing workloads versus de-identified or synthetic data only.
3. **Configuration governance.** Maintain a version-controlled configuration profile (e.g. JSON or YAML) that encodes the privacy modes, allowed providers, and default de-ID flags for the institution. Changes to this profile should follow a documented approval process involving clinical leadership, information security, and, where applicable, data protection officers.
4. **Endpoint hardening.** Ensure that the machines used for SynapseCore are protected by full-disk encryption, up-to-date operating system patches, and appropriate endpoint

detection and response tools. This is especially critical for psychiatric services, where clinicians may use laptops that travel between wards, clinics, and home offices.

5. **Backup and retention.** Since the reference build does not store PHI by default, backup responsibilities primarily concern exported files and any integration with local EHR systems. Institutions should define retention and destruction policies for such exports in line with broader medical record regulations.

From a psychiatric safety perspective, it is essential to document that SynapseCore is not an EHR and does not constitute the system of record. It is a workbench that facilitates note drafting, measurement-based care, and AI-assisted reasoning; the canonical clinical documentation must still reside in the institution's official record-keeping system, which will have its own privacy and security controls.

16.2.8 No Raw PHI Leaving the Local Environment Without Explicit Consent

Finally, we make explicit a central design and governance principle:

> *No raw PHI should leave the local environment via AI or logging routes unless a licensed clinician or authorised institution has explicitly enabled such behaviour, with a clear legal and ethical basis and appropriate contractual and technical safeguards.*

Operationalising this principle in SynapseCore involves multiple layers:

- Default privacy modes that disable PHI-bearing outbound calls.
- UI affordances that make privacy-relevant decisions visible (e.g. clear labelling of AI routes as "local only" vs. "cloud API" and warnings when PHI-containing content is about to be sent).
- Configuration and audit trails that record when privacy modes are changed, by whom, and under what justification.
- Institutional policies that define when and how de-identified versus identified data may be used with cloud AI providers, consistent with standard clinical ethics guidance on digital tools in psychiatry.(Torous et al., 2018)

In summary, the data privacy and security model of SynapseCore is intentionally conservative, privileging clinician control, client-side confinement of sensitive state, and de-identification by default. At the same time, the architecture remains flexible enough to support future extensions—for example, on-premise large language models and FHIR-based EHR integration—without weakening the core guarantees that psychiatric encounter data remain under the stewardship of the treating clinician and their institution.

16.3 Regulatory Pathways

This subsection synthesises how a SynapseCore-style digital psychiatry workbench fits within existing regulatory concepts for software in medicine, with emphasis on two questions: (i) under which conditions particular SynapseCore configurations could be considered a medical device (or *Software as a Medical Device*, SaMD) and (ii) what forms of clinical evaluation and documentation would be required if a given deployment were brought forward for regulatory clearance or conformity assessment. We focus on the United States Food and Drug Administration (FDA) framework for clinical decision support (CDS) and SaMD, the International Medical Device Regulators Forum (IMDRF) SaMD guidance,(International Medical Device Regulators Forum, 2013, 2017) and the European Union Medical Device Regulation (MDR) 2017/745,(European Parliament and Council of the European Union, 2017) while keeping in

view general ethical discussions around AI in psychiatry and mental health care.(Torous et al., 2018; Vayena et al., 2018)

16.3.1 Positioning Relative to Medical Device Definitions

At a high level, both FDA and EU MDR definitions hinge on whether software is intended by the manufacturer to be used for one or more *medical purposes* such as the diagnosis, prevention, monitoring, prediction, prognosis, treatment, or alleviation of disease.(European Parliament and Council of the European Union, 2017; International Medical Device Regulators Forum, 2013) For SynapseCore, this intent is encoded not only in narrative descriptions but also in which modules are enabled, how they are configured, and how they are integrated with the institutional record-of-care (e.g. the EHR).

Let $\mathcal{F}$ denote the set of distinct SynapseCore feature modules:

$$\mathcal{F} = \{f_{\text{notes}}, f_{\text{flows}}, f_{\text{mbc}}, f_{\text{registry}}, f_{\text{ai-doc}}, f_{\text{ai-consult}}, f_{\text{timer}}, f_{\text{export}}, \dots\},$$

where, for example, f_{flows} captures structured safety and capacity flows (e.g. agitation, suicidality, observation), f_{mbc} denotes the measurement-based care calculators and score deltas (e.g. PHQ-9, GAD-7, BFCRS), and f_{registry} refers to the cohort registry and risk gradings (G1–G5).

We define a mapping

$$c : \mathcal{F} \longrightarrow \{0, 1\}, \tag{16.9}$$

where $c(f) = 1$ indicates that feature f is configured and described in such a way that it meets the regulator's definition of a medical device function (e.g. it provides a recommendation intended to be used directly for diagnosis, treatment selection, or safety-critical decisions), whereas $c(f) = 0$ denotes non-device functionality (e.g. educational simulation, documentation assistance, or general-purpose IDE-like tooling).

Under this abstraction, a *deployment configuration* κ is a subset $\kappa \subseteq \mathcal{F}$ together with an *intended use statement* $U(\kappa)$ (e.g. "educational use in a residency programme", "prototype for research on risk stratification", or "clinical decision support for acute agitation in the emergency department"). The overall device classification predicate is then:

$$\mathsf{MD}(\kappa) = \begin{cases} 1, & \text{if } \exists f \in \kappa \text{ with } c(f) = 1 \text{ and } U(\kappa) \text{ is clinical,} \\ 0, & \text{otherwise.} \end{cases} \tag{16.10}$$

Thus, the same codebase can be non-device (e.g. in a sandbox mode with synthetic data and explicit educational intent) or device-like (e.g. when configured as a safety-critical CDS tool integrated into the EHR) depending on how features and intended use are combined.

Table 77 illustrates how major SynapseCore modules can be positioned along this spectrum. These entries are not regulatory determinations but design targets that inform how we engineer transparency, overrideability, and documentation.

In psychiatric practice, arguably the most sensitive axis is whether the system is used to *drive* safety-critical actions (e.g. involuntary hospitalisation, seclusion, rapid tranquillisation) versus simply structuring the reasoning that a human clinician must still own. We design SynapseCore so that, by default, it falls on the latter side: it structures documentation, exposes measurement-based care signals, and orchestrates AI tools whose outputs are always intended as drafts and suggestions, not automated orders.

16.3.2 Mapping to SaMD Risk Categories

The IMDRF SaMD framework characterises risk based on two dimensions: (i) the *significance* of the information provided by the SaMD to the healthcare decision (inform, drive, or treat/diagnose) and (ii) the *state of the healthcare situation or condition* (non-serious, serious, or critical).(International Medical Device Regulators Forum, 2013) We can represent each SynapseCore clinical function as a tuple

$$\phi = (\sigma, \theta),$$

where $\sigma \in \{\text{inform}, \text{drive}, \text{treat/diagnose}\}$ and $\theta \in \{\text{non-serious}, \text{serious}, \text{critical}\}$. A mapping

$$R : \Phi \longrightarrow \{\text{SaMD I}, \text{SaMD II}, \text{SaMD III}, \text{SaMD IV}\}$$

then assigns an IMDRF risk category to each function ϕ, where higher categories correspond to higher regulatory scrutiny.

For example, consider a suicide risk documentation flow that: (i) supports the clinician in capturing ideation, intent, means, and protective factors, (ii) generates a narrative summary, and (iii) displays a non-binding risk grade G1–G5 with an accompanying tooltip and colour class (cf. `riskGrades.ts`). If the grade is transparently derived from observable inputs and the narrative explicitly states that the grade is a *summary of clinician-entered information*, not an automated prediction, we target $\sigma = \text{inform}$. The condition θ may still be *critical* (e.g. acute suicidality), but because the function informs rather than drives or executes treatment, it stays in a lower SaMD category.

By contrast, if a future module were to compute a machine-learned risk score $s \in [0, 1]$ for self-harm or hospitalisation based on registry data, and then auto-prioritise patients above a threshold τ for outreach, we would have:

$$\sigma = \text{drive}, \qquad \theta \in \{\text{serious}, \text{critical}\},$$

and the regulatory category would likely be higher (e.g. SaMD III or IV depending on exact use). The same Random Forest or neural network code, when re-purposed for retrospective research in a fully de-identified sandbox, could instead be treated as non-device research software, reinforcing the importance of clearly articulating $U(\kappa)$ and $R(\phi)$ for each deployment.

16.3.3 Clinical Evaluation and Trial Design

If a SynapseCore configuration is intended to be brought forward as SaMD, regulators expect a structured clinical evaluation. The IMDRF framework and related FDA guidance outline three pillars:(International Medical Device Regulators Forum, 2017; U.S. Food and Drug Administration, 2017) (i) valid clinical association, (ii) analytical (technical) validation, and (iii) clinical validation.

Valid clinical association. We must justify that the input–output relationship encoded in the software corresponds to an accepted clinical construct. For example, scoring PHQ-9 and interpreting thresholds for depression severity relies on established psychometric validation.(Kroenke et al., 2001b) Similarly, BFCRS (Bush-Francis Catatonia Rating Scale) scores used in catatonia flows must respect consensus thresholds.(Bush et al., 1996) In SynapseCore, we make this explicit by: (a) naming the scales and severity bands directly in code (e.g. `AssessmentKind = "PHQ9" | "GAD7" | "BFCRS"` and band mappings), (b) linking each mapping back to the original validation references in the knowledge base, and (c) exposing these assumptions in documentation and UI tooltips.

Analytical validation. Analytical validation concerns whether the software implementation correctly and reliably executes its intended transformations. For scale scoring functions S, this translates into verifying that, for each item response vector x,

$$S(x) = \sum_{i=1}^{n} w_i x_i,$$

with the correct weights w_i and bounds, and that score bands are correctly mapped to severity labels (e.g. "mild", "moderate", "severe"). For AI-based modules, analytical validation includes regression tests on standardised prompt–response pairs and monitoring of distributional properties (e.g. token counts, hallucination flags).

More generally, we can define an analytical error functional

$$E_{\text{analytical}} = \mathbb{E}\big[\mathbf{1}\{S_{\text{impl}}(X) \neq S_{\text{spec}}(X)\}\big],$$

where S_{impl} is the implemented scoring or model function, S_{spec} is the specification, X is a representative random input, and $\mathbf{1}\{\cdot\}$ is the indicator function. The engineering goal is to drive $E_{\text{analytical}}$ as close to zero as feasible via unit tests, property-based testing, and continuous integration.

Clinical validation. Clinical validation addresses whether the use of the software in actual care settings leads to improved or at least non-inferior outcomes compared to usual care. In psychiatry, relevant endpoints include: change in symptom scores (e.g. PHQ-9, GAD-7), remission rates, hospitalisation and readmission rates, time to treatment response, and safety outcomes (e.g. suicide attempts, episodes of severe agitation requiring coercive measures).(Fortney et al., 2017b)

Suppose we consider a randomised evaluation of a SynapseCore-supported workflow versus standard care. Let $Y \in \mathbb{R}$ denote a continuous outcome (e.g. change in PHQ-9 score at 12 weeks) and $Z \in \{0, 1\}$ indicate whether SynapseCore-supported care was used. A simple superiority trial might test the null hypothesis

$$H_0 : \mathbb{E}[Y \mid Z = 1] - \mathbb{E}[Y \mid Z = 0] = 0$$

against a one-sided alternative $\mathbb{E}[Y \mid Z = 1] - \mathbb{E}[Y \mid Z = 0] < 0$ (i.e. greater symptom reduction in the SynapseCore arm). For non-inferiority designs in safety-sensitive contexts (e.g. self-harm outcomes), the null might instead be

$$H_0 : \mathbb{E}[Y \mid Z = 1] - \mathbb{E}[Y \mid Z = 0] \geq \Delta,$$

for some clinically acceptable margin $\Delta > 0$.

Risk-stratified analyses are also natural given the registry structure and risk grades. If $G \in \{1, \ldots, 5\}$ is the baseline risk grade, we can model outcomes via:

$$\mathbb{E}[Y \mid Z, G] = \alpha + \beta Z + \gamma G + \delta (Z \cdot G),$$

to assess whether SynapseCore's impact varies by baseline risk.

Pragmatic trial designs. Because SynapseCore is designed as a workbench rather than a single monolithic intervention, stepped-wedge or cluster-randomised designs at the team or ward level may be appropriate.(Hussey & Hughes, 2007) This allows entire clinical teams to adopt the tool, reducing contamination, while still enabling robust comparisons over time. Embedded analytics (via local logging of metadata, as described in Section 16.2) can then support secondary analyses of fidelity, usage patterns, and learning curves without compromising privacy.

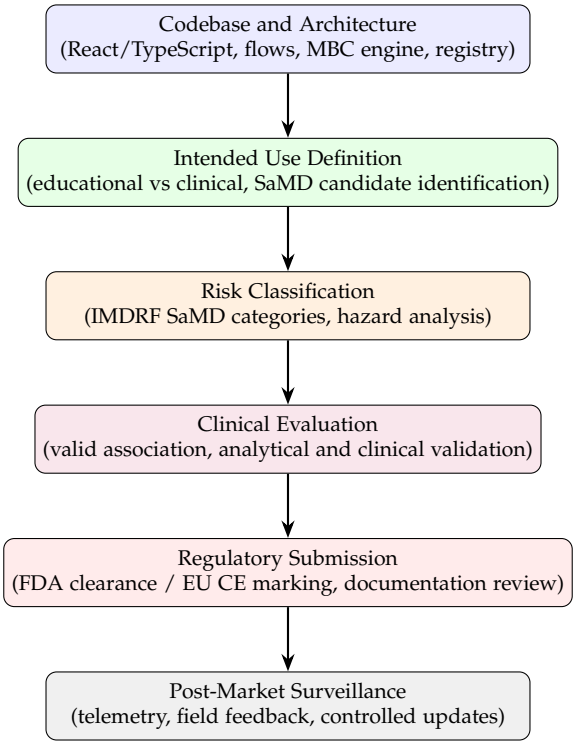


Figure 53: *Vertically structured regulatory pathway for a SynapseCore-based SaMD configuration. The pathway ascends from the technical codebase and architecture, through intended use, risk classification, and clinical evaluation, to regulatory submission and ongoing surveillance.*

16.3.4 Documentation and Regulatory Artefacts

A SaMD-capable SynapseCore build would require a comprehensive documentation suite aligned with relevant standards (e.g. ISO 13485 for quality management, IEC 62304 for software lifecycle, and IEC 62366 for usability engineering). Table 78 summarises key artefacts and how they map onto the existing architecture.

From a practical perspective, much of the groundwork for these artefacts is already implicit in the strongly typed React/TypeScript codebase and the modular architecture. However, to satisfy regulators, this information must be exported into durable, reviewable documents with traceability matrices linking requirements to code modules, test cases, and risk controls.

16.3.5 Vertically Organised Regulatory Pathway Schema

Figure 53 presents a vertically arranged, colour-coded schema of the regulatory pathway for a SynapseCore-based SaMD candidate. Each layer is constrained to a moderate width so that the figure respects page margins while remaining legible.

The intention is that the same foundational code can support multiple pathways: some configurations will terminate at the "Intended Use Definition" layer as non-device educational or research tools; others, with appropriate risk classification and evaluation, can progress to full SaMD status.

16.3.6 Outlook for Learning Systems in Psychiatry

Regulators are increasingly confronted with AI/ML-based SaMD that may adapt over time.(U.S. Food and Drug Administration, 2019; Vayena et al., 2018) SynapseCore is architected to separate: (i) the *AI orchestration layer* (model selection, routing, sampling), (ii) the *clinical content and workflows* (flows, registry, MBC), and (iii) the *observability and configuration* (telemetry, configuration stores, de-identification). From a regulatory standpoint, this separation supports the notion of a *predetermined change control plan* (PCCP) in which certain classes of changes (e.g. swapping one LLM for another within pre-specified performance bounds, or updating psychometric mapping tables based on new guidelines) can be pre-authorised without requiring an entirely new submission.

Let θ denote the parameters of a machine-learned component (e.g. weights of a risk prediction model) and $\mathcal{C}$ denote the space of allowed configurations. A PCCP-like abstraction partitions the change space into:

$$\Delta\theta = \underbrace{\Delta\theta_{\text{within}}}_{\text{permitted, auto-managed}} \oplus \underbrace{\Delta\theta_{\text{outside}}}_{\text{requires new review}},$$

and similarly for configuration changes $\Delta\mathcal{C}$. SynapseCore's configuration stores and model registry make such partitions explicit at the code level (e.g. which models are "experimental" vs "clinically approved"), and its metadata logging supports the monitoring required to detect drift or performance degradation.

For psychiatry, where outcomes are multidimensional and often long-horizon, this regulatory adaptability is crucial. It allows systems like SynapseCore to evolve as evidence accumulates—for example, refining how registry-based risk grades are computed or reweighting protective factors in safety flows—while maintaining rigorous oversight and auditability. In turn, this creates a path from the current status of SynapseCore collaboration with regulators, patients, and professional bodies.

Table 77: *Illustrative mapping of SynapseCore feature modules to regulatory categories under typical configurations.*

Feature module	Typical functionality	Regulatory stance (default)	Conditions for SaMD-like classification
Notes and IDE shell (f_{notes})	Free-text documentation, code-like editor for encounter notes, templates, and exports.	Non-device (productivity / documentation tool).	When used as the primary interface for generating structured orders or clinical instructions transmitted into the EHR.
Structured flows (f_{flows})	Safety and capacity flows that generate narrative summaries from structured inputs (e.g. suicidality, agitation, observation).	Low-risk CDS, potentially non-device if clinician can independently review basis.(U.S. Food and Drug Administration, 2022b)	If flows embed prescriptive treatment algorithms (e.g. auto-suggesting involuntary hold vs discharge) beyond transparent documentation.
MBC calculators (f_{mbc})	Scoring of PHQ-9, GAD-7, BFCRS, and computation of deltas; visualisation of change over time.	Device-like when scores are used to drive treatment pathways or safety flags.	When scores trigger automated recommendations (e.g. antidepressant dose changes) rather than manual clinician interpretation.(International Medical Device Regulators Forum, 2017)
Registry and risk grades (f_{registry})	Cohort view with risk grades (G1–G5), tags (e.g. SUD, FEP), and filters.	CDS / quality improvement dashboard; classification depends on how risk grades are generated and used.	When risk models systematically drive prioritisation or outreach workflows in a way that affects care access.
AI-assisted documentation ($f_{\text{ai-doc}}$)	Drafting and summarisation of narrative content (e.g. encounter notes, letters) based on clinician-entered text.	Generally non-device or low-risk CDS if output is transparently editable and clearly labelled as a draft.	If used to auto-populate critical sections (e.g. medication plans) without clinician review, classification may shift towards device.
AI consult tools ($f_{\text{ai-consult}}$)	Natural-language consulton tools for supervision-like reasoning, literature summaries, or reflective analysis.	Educational / research tool by default; intended to support, not replace, expert judgement.	When deployed as a clinical decision engine with direct treatment recommendations.

Table 78: *Core regulatory documents and their relationship to SynapseCore architecture and code.*

Document	Content focus	SynapseCore anchors
Intended Use and Indications for Use	Concise statement of medical purpose, target population, and care setting.	Section 16.1; configuration profiles for flows, MBC, and registry features; toggles indicating educational vs clinical modes.
Software Requirements Specification (SRS)	Functional and non-functional requirements, including safety, performance, and interoperability.	TypeScript type definitions (e.g. registry `types.ts`), flow schemas, AI route configuration; mapping to user stories.
Software Architecture Description	High-level module diagrams and data flows, including AI orchestration and de-identification.	AI provider registry, sampling mapper, centerpanel shells, de-identification and logging stack (Sections 14–16).
Risk Management File (ISO 14971)	Hazard identification, risk estimations, mitigations, and residual risk evaluation.	Structured safety flows (e.g. suicidality, catatonia); design decisions about defaults, warnings, and overrideability; risk grades G1–G5.
Verification and Validation (V&V) Plan and Reports	Test strategy, unit and integration tests, simulation runs, and clinical study reports.	Test suites for scoring functions and flows; simulated registries; clinical trial protocols and results.
Usability Engineering File (IEC 62366)	Human factors analyses, formative and summative usability tests, and UI risk controls.	Center panel layout; risk grade visual encoding (colours, labels, tooltips); error messaging and confirmation flows.
Cybersecurity and Privacy Documentation	Threat models, de-identification strategy, logging design, and network boundary controls.	Client-only storage, privacy modes, AI egress policy (Section 16.2); deployment hardening guidelines.
Post-Market Surveillance and Change Control Plan	Monitoring strategy, field feedback processes, and change classification (e.g. PCCP for learning systems).(U.S. Food and Drug Administration, 2019)	Telemetry aggregation (for de-identified metrics), model versioning, and configuration management for AI routes and flows.

17 Limitations

The present SynapseCore implementation is intentionally scoped as a browser-based, front-end–only workbench for digital psychiatry and clinical reasoning. This architectural decision was driven by a desire to maximise privacy (client-side confinement of sensitive data), ease of self-hosting, and experimental flexibility for psychiatrists who wish to explore structured flows, measurement-based care (MBC), and AI-assisted documentation without installing heavy server components or modifying institutional electronic health record (EHR) systems.(Torous et al., 2018) At the same time, this design creates systematic constraints that must be acknowledged when interpreting the capabilities and generalisability of the tool.

Conceptually, the system can be decomposed into four interacting layers,

$$\mathcal{S} = \mathcal{U} \circ \mathcal{F} \circ \mathcal{A} \circ \mathcal{D},$$

where $\mathcal{U}$ denotes the user interface and interaction layer (React/TypeScript shell, centre panel flows, timers), $\mathcal{F}$ the psychiatric domain layer (safety and capacity flows, MBC scoring, registry views), $\mathcal{A}$ the AI orchestration and prompting subsystem, and $\mathcal{D}$ the data and telemetry layer (in-memory state, de-identification transforms, local metadata logging). The strengths and weaknesses of the current build can be understood as functions of constraints at each of these levels.

For clarity, we distinguish three broad families of limitations:

1. *Technical limitations*, reflecting architectural choices (browser-only execution, lack of direct EHR integration, dependence on external AI providers, finite testing coverage).
2. *Clinical limitations*, such as the range of psychiatric conditions and care settings explicitly modelled, and the current level of clinical evidence for improved outcomes.(Fortney et al., 2017b)
3. *User and context limitations*, including training requirements, local infrastructure, and organisational culture.

The present subsection focuses on the first category, because technical constraints shape what is currently feasible and what kinds of future integration work will be needed before SynapseCore can be considered for routine deployment or regulatory evaluation as Software as a Medical Device (SaMD).(International Medical Device Regulators Forum, 2013, 2017)

17.1 Technical Limitations

17.1.1 Browser-only architecture and lack of backend EHR integration

The most fundamental limitation is that SynapseCore presently runs entirely in the clinician's browser. All clinically meaningful state—including encounter notes, structured flow answers, and scale responses—resides in the React component tree and related stores. There is no dedicated SynapseCore server, database, or message bus. Any integration with the institutional EHR occurs indirectly through manual copy/paste of summaries or through exports that the clinician then imports into the EHR using existing workflows.

Let $R_t(i)$ denote the official EHR record for patient i at clinical time t, and $S_t(i)$ the SynapseCore working state (notes, flows, scores) at the same time. In the current architecture there is no programmatic synchronisation map $\Psi : S_t(i) \to R_t(i)$ or $\Phi : R_t(i) \to S_t(i)$ enforced by the system. Instead, the relation between these states is mediated by the clinician:

$$R_t(i) = \mathsf{EHR\_update}\big(R_{t^-}(i),\, \mathrm{copy}(\mathsf{export}(S_t(i)))\big),$$

where export denotes SynapseCore's summary/export functions and EHR_update the clinician's manual update of the record. Any failure in this human-mediated step (e.g. incomplete transfer, incorrect patient, or copy/paste errors) can lead to discrepancies $\Delta_t(i) = R_t(i) - S_t(i)$ between the workbench and the juridical record-of-care.

This has several consequences:

- **No system-of-record role.** SynapseCore is not an EHR and cannot, in its present form, assume the legal responsibilities of record-keeping. It is a reasoning and documentation assistant whose outputs must ultimately be reconciled with the EHR, which remains the authoritative source.
- **Limited longitudinal persistence.** Repeated use of MBC scales (PHQ-9, GAD-7, BFCRS) within SynapseCore enables the computation of intra-session and short-horizon deltas, but true longitudinal analysis across months or years requires a persistent patient time series $\{Y_{i,t}\}_t$ stored in a database or EHR.(Bush et al., 1996; Kroenke et al., 2001b) Without such a backend, the tool cannot guarantee comprehensive cohort-level coverage or survivorship-aware analyses.
- **Limited multi-user and team workflows.** In multi-disciplinary teams, different clinicians may see a given patient at different times. Because SynapseCore stores working state per browser instance, there is no built-in mechanism for team members to share a canonical "SynapseCore view" of a patient in real time; such coordination must occur through the EHR or other channels.

From the perspective of interoperability standards such as HL7 FHIR, the current implementation effectively operates as an external client without a registered identity or dedicated backend FHIR gateway.(Mandel et al., 2016) Future work will be required to design optional, institution-owned adapters that map SynapseCore flows and summaries onto FHIR resources (e.g. `Observation`, `QuestionnaireResponse`, `CarePlan`) while preserving the privacy guarantees described in Section 16.2.

17.1.2 Multi-device use and absence of seamless state portability

The browser-only model also constrains how clinicians can move between devices. Consider a clinician who sometimes works on a hospital desktop and sometimes on a laptop. Let $S_t^{(d)}$ denote the SynapseCore state on device d at time t. In the absence of a shared backend or secure cloud store, there is no automatic synchronisation, so

$$S_t^{(d_1)} \neq S_t^{(d_2)} \quad \text{for generic } t,$$

unless the clinician manually transfers exported files between devices and re-imports them.

Practically, this means:

- SynapseCore behaves more like a sophisticated local notebook or IDE than like a cross-device productivity suite.
- Loss of a device (e.g. stolen laptop) may lead to loss of unsaved SynapseCore work, although critical information should already reside in the EHR; nonetheless, the lack of built-in encrypted cloud backup is a limitation compared to mainstream enterprise software.
- Institutional policies that mandate central audit of all clinical tools may find it more challenging to track SynapseCore usage when state is fragmented across multiple endpoints.

While this constraint is partly a deliberate privacy-preserving choice, it reduces the con-

venience of SynapseCore for clinicians accustomed to cloud-synchronised applications and will need to be addressed—for instance via optional, institution-managed encrypted sync services—if the tool is to be adopted at scale.

17.1.3 Dependence on external AI providers and model drift

The AI orchestration layer in SynapseCore is explicitly provider-agnostic: it can route prompts to different large language model (LLM) endpoints via a configurable registry. However, in the reference implementation these endpoints are external services, and SynapseCore has no control over their training data, update schedules, or internal safety systems.

Formally, each AI call can be viewed as an evaluation of a stochastic function

$$Y = f_{\theta(t)}(X, C, U),$$

where X is the prompt (containing transformed clinical or synthetic data), C the configuration (e.g. temperature, max tokens, system prompt), U the upstream provider context (e.g. account, deployment region), and $\theta(t)$ the latent model parameters at time t. Even with the same (X, C, U), the output distribution $P(Y \mid X, C, U, t)$ may change over time due to:

- provider-driven model updates ($\theta(t_2) \neq \theta(t_1)$);
- changes in safety filters, content classifiers, or prompt routing; and
- modifications in infrastructure (e.g. load-balancing across model families).

For psychiatrically relevant use cases, this yields several technical limitations:

- **Non-reproducibility of AI-assisted narratives.** A consult-like explanation or note draft obtained at time t_1 cannot be reproduced exactly at time t_2, even with deterministic settings. This complicates regression testing, legal discovery, and academic replicability.(International Medical Device Regulators Forum, 2017; Vayena et al., 2018)
- **Uncertain failure modes.** External LLMs may produce rare but clinically significant hallucinations or biased statements, particularly in domains under-represented in training data (e.g. cultural formulations, gender-diverse identities, forensic psychiatry). Because we cannot inspect training corpora directly, the safety analysis must rely on black-box stress testing and conservative guardrails.
- **Policy and availability risk.** Provider outages, pricing changes, and acceptable-use policy updates can degrade functionality or necessitate rapid reconfiguration. For example, an endpoint previously approved for PHI processing may later restrict such usage, forcing institutions to switch to on-premise or alternative vendors.

From an engineering viewpoint, the reliability of an AI-powered workflow can be decomposed as

$$R_{\text{AI}} = R_{\text{client}} \cdot R_{\text{network}} \cdot R_{\text{provider}},$$

with R_{client} representing internal robustness (input validation, error handling, prompt construction), R_{network} the stability of connectivity, and R_{provider} the provider's availability and internal correctness. SynapseCore directly controls only R_{client}; the other factors are external, and their variation should be considered when designing clinical workflows that depend on AI output.

17.1.4 Local compute constraints and limited offline capability

Because execution is confined to the browser, SynapseCore is limited by the compute and memory resources available on typical clinician devices. Although modern browsers support

WebAssembly and WebGPU, enabling modest on-device inference, it remains infeasible to run very large LLMs or complex multimodal models entirely in-browser on standard hardware.

Let C_{req} denote the approximate computational requirement of a desired model (e.g. FLOPs per token) and C_{local} the effective per-session budget on the clinician's device. Whenever $C_{\text{req}} \gg C_{\text{local}}$, the system must delegate inference to remote endpoints, reinforcing the dependence described above. In low-bandwidth or intermittently connected settings—ironically, often those with the greatest unmet psychiatric need—AI-assisted features may become unavailable or degraded.(Torous et al., 2018)

Moreover, browser lifecycle rules introduce additional limitations:

- background tabs may be suspended, interrupting long-running tasks;
- aggressive memory reclamation by the browser can force reloading of in-memory state if pages are left inactive; and
- automatic browser or OS updates may alter performance characteristics without explicit notice.

SynapseCore mitigates some of these issues by ensuring that core flows and scoring functions (e.g. PHQ-9, GAD-7, BFCRS, suicidality flows) are fully functional without AI and by encouraging clinicians to export or commit important summaries promptly. However, full offline support remains beyond the scope of the current build.

17.1.5 Privacy-preserving logging versus observability

As detailed in Section 16.2, telemetry in SynapseCore is deliberately restricted to metadata (e.g. route identifiers, timing, token counts) and excludes raw clinical text. Formally, if C is the space of clinical texts and E the space of telemetry events, the logging map $\ell : C \to E$ satisfies

$$\forall c \in C : \quad \text{content}\big(\ell(c)\big) \cap C = \varnothing.$$

While this approach significantly reduces privacy risk and simplifies self-hosted deployments, it introduces several technical trade-offs:

- **Limited forensic reconstruction.** In the event of a suspected AI-related incident (e.g. inappropriate suggestion, harmful phrasing), we cannot reconstruct the exact prompt and response from logs alone; reconstruction depends on clinician-provided examples (screenshots, copied text) or on synthetic replication attempts.
- **Constrained large-scale error analysis.** Many modern ML safety improvements rely on mining massive real-world interaction logs to find rare but concerning patterns. With no clinical text in telemetry, SynapseCore cannot implement such pipelines without additional, explicitly consented data-collection studies under ethics oversight.(Vayena et al., 2018)
- **Challenges for continuous learning.** Because the system does not retain encounter-level text, any fine-tuning or adaptation of models to local populations must be performed using separately curated datasets stored under appropriate governance structures, outside the core application.

Thus, the strong privacy posture creates what might be termed a *validation and improvement bottleneck*: we retain theoretical and bench-level assurance for specific components (e.g. scoring algorithms, flow transitions), but we cannot automatically harvest the full statistical signal of routine clinical use to drive rapid model refinement.

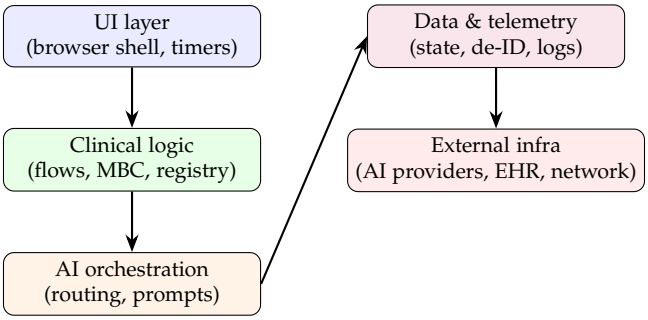


Figure 54: *Two-column schematic of the main technical layers in SynapseCore. The left column shows core application layers; the right column shows data/telemetry and external infrastructure on which the system depends.*

17.1.6 Configuration space and finite testing coverage

A further technical limitation arises from the size of SynapseCore's configuration space. Clinicians or institutions can customise:

- which flows are enabled (e.g. suicidality, catatonia, capacity, agitation, observation);
- which scales and scoring rules are active for a given population (e.g. PHQ-9 vs PHQ-8, GAD-7 vs GAD-2);(Kroenke et al., 2001b)
- which AI routes and providers are available in each privacy mode; and
- how risk grades and visual encodings (colours, tooltips) are mapped.

If we denote the number of choices for each configurable dimension by $n_1, n_2, \ldots, n_k$, the total configuration space has size $|\mathcal{C}| = \prod_{j=1}^{k} n_j$, which can easily become large even when each n_j is modest. It is infeasible to test all configurations exhaustively.

Instead, we rely on:

- strong static typing (TypeScript) to rule out wide classes of structurally invalid configurations at compile time;
- unit tests for pure functions such as scoring and flow reducers to ensure analytical correctness across representative inputs; and
- curated "canonical" configuration profiles (e.g. adult outpatient, emergency psychiatry, consultation-liaison) that receive scenario-based integration testing and manual expert review.

Let T denote the set of test cases and $\pi : \mathcal{C} \to \{0, 1\}$ an indicator that a configuration is covered by some test in T. The effective coverage is then

$$\mathrm{Cov}(T) = \frac{1}{|\mathcal{C}|} \sum_{c \in \mathcal{C}} \pi(c),$$

which is necessarily less than one. The purpose of the risk-management and post-market surveillance processes (Section 16.3) is to ensure that high-risk regions of $\mathcal{C}$ receive disproportionately high attention and that field feedback drives targeted expansion of T over time.(U.S. Food and Drug Administration, 2019)

Table 79: *Synopsis of major technical limitations in the current SynapseCore build, with clinical impact and mitigation strategies.*

Limitation	Concrete manifestation	Clinical impact	Current / planned mitigation
Browser-only architecture, no backend EHR	No programmatic synchronisation between SynapseCore and the institutional system of record; no central cohort store.	Clinicians must manually transfer summaries; risk of divergence between workbench and EHR; limited longitudinal analytics.	Explicit positioning as a workbench; robust export functions; design of optional FHIR-based adapters as institution-owned components.(Mandel et al., 2016)
Multi-device fragmentation of state	Separate local state on each device; no automatic cross-device sync.	Reduced convenience for clinicians moving across workstations; potential loss of unsaved work if a device fails.	Encourage rapid export of key artefacts; future exploration of encrypted, institution-managed sync for approved deployments.
Dependence on external AI providers and model drift	Hidden model updates, outages, and policy changes affect AI behaviour.	Variability in AI-assisted narratives and consult-style outputs; need for clinician vigilance and conservative reliance.	Provider-agnostic routing; conservative prompts; ability to disable or re-route AI tools; emphasis on human-in-the-loop use.(Vayena et al., 2018)
Local compute and connectivity constraints	Heavy models cannot run fully in-browser; AI features degrade offline.	Limited AI support in low-bandwidth or resource-constrained settings; inequality of access to advanced features.	Core flows and scoring implemented without AI; fallbacks for network errors; future evaluation of lightweight on-device models.
Privacy-preserving logging reduces observability	Telemetry excludes clinical text, restricting comprehensive log analysis.	Harder to detect rare AI failure modes or subtle biases; limited automatic continuous learning.	Bench-level stress tests; ethics-approved research deployments with explicit consent; structured incident reporting.(Huckvale et al., 2019)
Large configuration space vs finite tests	Many possible combinations of flows, scales, privacy modes, and AI routes.	Rare configuration combinations may interact in unanticipated ways.	Strong typing; canonical profiles; risk-based test prioritisation; post-market feedback feeding into expanded test suites.(U.S. Food and Drug Administration, 2019)

17.2 Clinical Limitations

The clinical footprint of the present SynapseCore build reflects a set of deliberate priorities rather than an attempt at encyclopaedic coverage. The codebase and content library have been optimised around measurement-based care (MBC) for common mood and anxiety disorders, high-stakes risk and safety formulations, catatonia and behavioural disturbance flows, and consultation-liaison style encounters.(Fortney et al., 2017a; Ko et al., 2023; Lewis et al., 2019) By contrast, complex multi-morbid trajectories, rare syndromes, and certain specialist service lines are only sparsely represented or not modelled at all. In addition, SynapseCore has, at this stage, no large-scale prospective clinical evaluation as an intervention or decision-support adjunct; its clinical profile is therefore best understood as hypothesis-generating and exploratory rather than definitive.(American Psychiatric Association, 2021; Kilbourne et al., 2018)

To make these limitations explicit, we again work with an abstract coverage space

$$\mathcal{C} = \mathcal{D} \times \mathcal{S} \times \mathcal{P},$$

where $\mathcal{D}$ denotes diagnostic or problem domains, $\mathcal{S}$ care settings, and $\mathcal{P}$ population strata. Each point $c \in \mathcal{C}$ represents a triplet (d, s, p) of diagnosis (or presenting cluster), setting, and population context. We define a coverage function

$$q : \mathcal{C} \to [0, 1],$$

which heuristically quantifies how well a given configuration is supported by the current implementation in terms of *structured assessment, content, and intended use*. Values near $q(c) = 1$ indicate strong alignment (presence of dedicated flows, scale logic, exports, and examples), whereas values near $q(c) = 0$ indicate absence or only generic scaffolding.

For the present version we can, informally, identify a "core" clinical region $\Omega_{\text{core}} \subset \mathcal{C}$ where $q(c) \geq q_{\min}$ for some threshold $q_{\min} \in (0, 1)$, and an "under-modelled" complement $\mathcal{C} \setminus \Omega_{\text{core}}$. The following subsubsections describe how Ω_{core} is currently structured and, equally importantly, which areas of $\mathcal{C}$ remain insufficiently captured.

17.2.1 Scope of diagnostic coverage

At the diagnostic level, SynapseCore has been implemented with a clear emphasis on:

- high-prevalence mood and anxiety disorders, where robust MBC instruments (e.g. PHQ-9, GAD-7) and severity band cut-offs are available and already integrated into routine care in many systems;(Ko et al., 2023; Kroenke et al., 2001b; Lewis et al., 2019)
- trauma-related presentations, for which structured risk formulation, psychoeducation, and functional assessment can be scaffolded by generic content cards and flows; and
- catatonia, agitation, and acute behavioural disturbance, supported by dedicated flows and outcome logic in the codebase (e.g. catatonia and agitation outcome modules in the psychiatry feature tree).

This yields a relatively well-defined subset $\mathcal{D}_{\text{focus}} \subset \mathcal{D}$ in which the combination of scale logic, flows, and content is coherent. However, several important diagnostic families are only weakly represented:

- bipolar spectrum disorders and complex mood instability, where depression-focused instruments such as PHQ-9 are necessary but not sufficient, and where distinct longitudinal trajectories and risk patterns are critical;(Miklowitz et al., 2003)

- primary psychotic disorders and schizophrenia spectrum conditions, which demand structured assessment of positive, negative, and cognitive symptoms, as well as nuanced risk formulations and capacity assessments;
- eating disorders and feeding-related conditions, where highly specific psychopathology (e.g. body image disturbance, compensatory behaviours) and nutritional risk must be captured;
- substance use disorders and dual-diagnosis configurations, where craving, withdrawal, relapse risk, and social determinants play central roles; and
- personality disorders and chronic relational or attachment patterns, which require careful longitudinal formulation rather than visit-level snapshots.(Dixon et al., 2003; Fiorillo et al., 2015)

In terms of the coverage function q, we can think of $\mathcal{D}_{\text{focus}}$ as the subset where $q(d, s, p)$ is primarily limited by setting and population factors, whereas for the above domains we have structurally lower values of $q(d, s, p)$ across most s, p. Put differently, the current SynapseCore implementation does not aspire to be an exhaustive diagnostic workbench. It is, instead, deliberately concentrated on diagnostic niches where MBC and structured flows are reasonably mature, and where the literature suggests that closing the implementation gap may yield meaningful quality gains.(Fortney et al., 2017a; Kilbourne et al., 2018)

The practical implication is that SynapseCore should not be used in isolation to support primary diagnostic decision-making in complex, multi-axial presentations. Rather, it should be framed as a documentation and MBC-support layer on top of existing diagnostic workflows, with an explicit acknowledgement that certain diagnoses are essentially invisible to its current content and flow graph.

17.2.2 Settings, populations, and generalisability

The codebase and flows encode an implicit model of clinical workflow. In broad terms, the present version aligns most naturally with:

- adult outpatient consultations (initial assessments and follow-up visits);
- emergency and urgent risk assessments (via suicidality, agitation, and observation flows); and
- consultation-liaison psychiatry on general medical wards, where structured formulation and time-boxed interviews are typical.(Connolly et al., 2021)

There is, by contrast, limited explicit modelling of:

- inpatient ward workflows (e.g. ward rounds, observation reviews, seclusion/restraint documentation);
- community mental health team practice, including home treatment, crisis resolution, and assertive outreach;
- perinatal, forensic, and neuropsychiatric subspecialties, where different legal thresholds, multidisciplinary structures, and temporal rhythms apply; and
- primary care mental health integration pathways, where brief consultations and stepped-care models dominate.(Kilbourne et al., 2018)

Formally, if we view q as decomposed along setting and population axes,

$$q(d, s, p) = q_{\mathcal{D}}(d) \cdot q_{\mathcal{S}}(s) \cdot q_{\mathcal{P}}(p),$$

then $q_{\mathcal{S}}(s)$ is close to one for adult outpatient, emergency, and consultation-liaison settings, but approaches zero for unencoded settings (e.g. community crisis, forensic services). Similarly, the population factor $q_{\mathcal{P}}(p)$ is implicitly tuned to adults with at least moderate literacy and some familiarity with standard clinical encounters.

The current implementation has not been systematically evaluated in:

- children and adolescents, despite initial CAMHS seeds in the content library;
- older adults with poly-morbidity, frailty, and neurocognitive impairment;(Carlson & Yarns, 2023; Pati et al., 2021)
- individuals with significant sensory impairments or communication differences; or
- people from cultural and linguistic communities under-represented in the training data of upstream AI models.(American Psychiatric Association, 2021; Torous et al., 2018)

This imposes a clear generalisability limitation: even if SynapseCore were eventually shown to be beneficial in its core settings, such findings could not be naïvely extrapolated to under-represented contexts. Local adaptation—including content localisation, integration with culturally-specific guidelines, and co-design with service users and carers in those communities—would be necessary before deployment at scale.

17.2.3 Boundaries of decision-support and automation bias

By design, the SynapseCore workbench aims to stay on the descriptive side of decision support. The code avoids direct diagnostic or treatment recommendation engines and instead focuses on structuring information, computing scores, and generating narratives that clinicians can review and edit. Nonetheless, the use of large language models (LLMs) introduces the risk of *automation bias*: the tendency to over-rely on automated aids, following their output even when it conflicts with the user's own judgement.(Goddard et al., 2012; Lyell & Coiera, 2017)

Let J denote the clinician's unaided decision (e.g. risk grade, suggested diagnostic formulation), Y the AI-assisted suggestion, and A the event that the clinician ultimately chooses to adopt the AI suggestion. Of special interest is the conditional probability

$$\Pr\big(A \mid J = \text{correct}, Y = \text{incorrect}\big),$$

which corresponds to cases where a correct human judgement is overridden by an incorrect automated recommendation. Systematic reviews of CDSS use indicate that this probability is non-zero and can be substantial in certain contexts, especially under high cognitive load, time pressure, and when the automated display is visually salient.(Abdelwanis et al., 2024; Goddard et al., 2012; Lyell & Coiera, 2017)

SynapseCore's current user interface offers only basic mitigations:

- AI outputs are visually separated from deterministic scores and structured data in the centre panel and notes;
- explicit labels and tooltips emphasise that AI narratives are drafts or suggestions, not authoritative conclusions; and
- export and copy mechanisms make it clear that clinicians remain responsible for editing and approving any text that enters the record.

However, the system does not yet include more advanced mitigation strategies, such as:

- structured prompts that explicitly ask clinicians to consider counter-arguments before accepting AI suggestions;
- interface patterns that require clinicians to commit to a preliminary formulation before AI output is revealed; or
- adaptive throttling of AI suggestions in domains where automation bias appears particularly likely or harmful.(Abdelwanis et al., 2024; Shortliffe & Sepúlveda, 2018)

From a clinical governance standpoint, this means that while SynapseCore is not designed to function as a high-autonomy decision-support system, the risk of automation bias cannot

be regarded as negligible. Any institutional deployment would need to incorporate training, documentation, and monitoring that explicitly address human factors and cognitive biases in AI use.

17.2.4 Evidence base and lack of large-scale clinical evaluation

To date, SynapseCore has not been evaluated in randomised controlled trials or large-scale implementation studies. Existing evidence is limited to:

- unit and integration tests for deterministic components (e.g. scoring functions, flow reducers);
- internal use with synthetic or de-identified cases by clinicians and informatics experts;
- informal A/B comparisons of different prompt templates and export formats; and
- qualitative feedback on usability, cognitive fit, and perceived value in early prototypes.(Connolly et al., 2021; Vrdoljak et al., 2024)

Let $\mathcal{E}_{\text{bench}}$ denote this internal, bench-level evidence set and $\mathcal{E}_{\text{clinical}}$ the set of high-quality clinical studies (e.g. cluster-randomised trials, stepped-wedge implementations, pre/post cohorts with robust confounding adjustment). At present,

$$\mathcal{E}_{\text{clinical}} = \varnothing,$$

so any claims about the effect of SynapseCore on clinical outcomes, quality of care, documentation burden, or safety must be considered speculative.

The broader digital mental health literature underscores why such evaluation is non-trivial. Systematic reviews highlight:

- high dropout and low sustained engagement with many mental health apps;(Torous et al., 2018, 2020)
- inconsistent reporting of harms and adverse events; and
- variable methodological quality with underpowered or short-duration studies.(American Psychiatric Association, 2021; Philippe et al., 2022)

SynapseCore differs from consumer-facing apps in that its primary users are clinicians rather than patients, and its goal is to influence professional workflow rather than directly deliver therapy. As such, appropriate outcome measures might include:

- documentation time per encounter;
- adoption of MBC (e.g. proportion of eligible visits with complete PHQ-9 and GAD-7 documentation);(Fortney et al., 2017a; Ko et al., 2023)
- quality of risk formulations (e.g. completeness and internal coherence, assessed via blinded rating scales);
- error rates in structured scoring (e.g. PHQ-9 mis-summation, mis-band assignment); and
- clinician-reported cognitive load and burnout proxies.(Kilbourne et al., 2018)

Designing and executing such studies will require partnerships with academic and service institutions, clear pre-registration of hypotheses, and careful attention to unintended consequences (e.g. over-documentation, new failure modes induced by AI suggestions).(American Psychiatric Association, 2021; Ridout et al., 2024) Until then, SynapseCore should be regarded as an investigational tool suitable for prototyping, education, and carefully governed early-adopter use, rather than a clinically validated digital health technology.

17.2.5 Training, expertise, and local adaptation

The effective use of SynapseCore presupposes a baseline of clinical expertise, digital literacy, and institutional readiness that cannot be guaranteed in all contexts. The design assumes that users are licensed mental health professionals (or supervised trainees) familiar with diagnostic frameworks, risk formulation, and psychopharmacology, and that they will be able to critically evaluate AI output when it is present.(American Psychiatric Association, 2021; Torous et al., 2018)

There are no built-in mechanisms for:

- assessing a user's competence or scope of practice;
- dynamically tailoring content to training level (e.g. specialist vs generalist, consultant vs trainee); or
- preventing users from applying tools designed for one population or setting to another (e.g. adult tools in child services).

Moreover, many aspects of safe psychiatric practice are deeply local:

- thresholds for involuntary admission, observation levels, and duty to warn/protect differ across jurisdictions;
- service configurations (e.g. availability of crisis teams, day hospitals, intensive outpatient programmes) vary across institutions; and
- documentation norms and medico-legal expectations are shaped by local policy and case law.(American Psychiatric Association, 2021; Bourgeois et al., 2019)

Let Λ denote the vector of local practice parameters (legal, organisational, cultural). In the current build, Λ is not explicitly modelled or encoded in the configuration schema. Instead, it is implicitly expected that institutions adopting SynapseCore will:

- customise content (e.g. risk phrasing, section templates) and exports to reflect local policy;
- restrict or enable specific flows, scales, and AI routes according to local governance; and
- embed the tool within broader training, supervision, and quality assurance processes.

Absent such adaptation, there is a non-trivial risk of misalignment between the workbench's assumptions and local practice. This can be conceptualised as a *local fit gap* $\Delta_\Lambda(c)$, which measures the discrepancy between the tool's implicit model and the actual requirements at configuration c. In regions of the coverage space where both $q(c)$ and $\Delta_\Lambda(c)$ are poorly characterised, conservative deployment (or non-deployment) is warranted.

17.2.6 Multi-morbidity, social complexity, and longitudinal trajectories

Real-world psychiatric practice is dominated by multi-morbidity and social complexity rather than single-index diagnoses.(Kilbourne et al., 2018; Pati et al., 2021) Patients frequently present with combinations of mood, anxiety, substance use, physical health problems, and social adversity (e.g. housing insecurity, trauma exposure). While SynapseCore's flows and content can be composed to describe complex presentations, the underlying model remains primarily additive and visit-centric.

In more formal terms, we can view a patient's clinical state at time t as a tuple

$$X_t = (D_t, M_t, S_t, C_t),$$

where D_t encodes diagnoses or syndromal clusters, M_t symptom measures and functional scores (e.g. PHQ-9, GAD-7, sleep, functioning), S_t social determinants (e.g. housing, employment, relationships), and C_t contextual factors (e.g. recent crises, changes in medication). The

Table 80: *Illustrative mapping between the broader clinical coverage space and the present SynapseCore implementation. The tool currently emphasises adult mood/anxiety disorders, acute risk, and consultation-liaison style work, with limited coverage of other domains and settings.*

Axis	Examples in broader space	Representation in current SynapseCore	Clinical gap / limitation
Diagnostic domains	Mood, anxiety, psychosis, bipolar, SUD, ED, PD, neurocognitive, neurodevelopmental	Strongest support for mood/anxiety (PHQ-9, GAD-7), trauma-related presentations, catatonia, agitation; partial support for neuropsychiatric liaison; generic templates elsewhere	Limited condition-specific workflows for psychosis, bipolar, eating, substance use, personality and neurodevelopmental disorders
Care settings	Adult/child outpatient, inpatient, ED, CL, forensic, community teams	Focus on adult outpatient, emergency/urgent risk, and CL-style assessments; seeds for CAMHS and group work	No comprehensive encoding of inpatient, community crisis, forensic or perinatal workflows
Populations	Children, adolescents, adults, older adults; diverse cultural and linguistic groups; neurodiversity	Default adult-centric language; early CAMHS seeds; no systematic tailoring for older adults, low literacy, or non-dominant language groups	Unknown performance in under-represented populations; requires local adaptation and evaluation
Evidence base	RCTs, implementation studies, safety monitoring for digital tools	Bench-level testing, internal use on synthetic/ de-identified cases; qualitative feedback only	No randomised or large-scale observational studies of SynapseCore itself; uncertain impact on outcomes, safety, and workload

current content library provides relatively rich hooks into M_t and selected aspects of D_t, but far fewer into S_t and C_t. Furthermore, the workflow is oriented towards single-encounter snapshots, with only limited support for longitudinal reasoning over time series $\{X_t\}_t$.

This has several clinical implications:

- interventions that depend critically on cumulative social risk (e.g. homelessness, domestic violence, immigration status) are not explicitly encoded in flows and may be under-emphasised in AI-generated narratives;
- longitudinal treatment planning (e.g. across months of care) is partially supported by repeated MBC scores, but not by longitudinal visualisations or trajectory modelling modules; and
- coordination across multiple services (e.g. addiction, social work, primary care) is out of scope for the current build.

Addressing these gaps will require not only additional content and flows but also architectural support for storing, visualising, and reasoning over multi-dimensional patient trajectories. Until then, SynapseCore should be conceptualised as a tool that enhances structured reasoning within single encounters and short-horizon episodes, rather than as a full longitudinal care management platform.

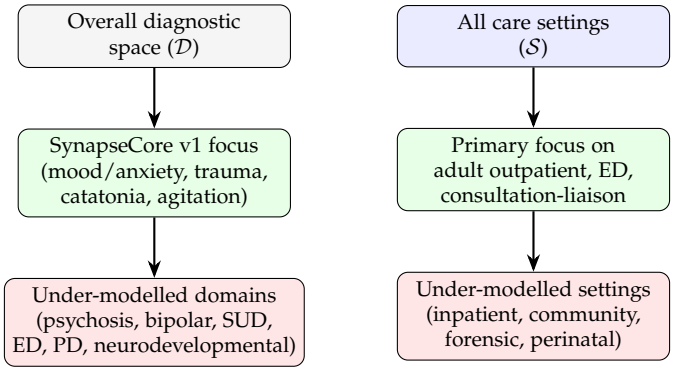


Figure 55: *Conceptual view of the clinical coverage space. The current SynapseCore implementation concentrates on a subset of diagnostic domains and care settings, leaving substantial under-modelled areas that require further design, localisation, and evaluation.*

17.3 User and Context Limitations

The current SynapseCore build is not only shaped by technical and clinical choices, but also by implicit assumptions about who its users are and in which contexts they work. These assumptions span language, digital literacy, hardware and network resources, institutional governance, and broader social determinants such as broadband availability. As with most digital mental health tools, these dimensions are not neutral: they can either amplify or attenuate inequities, depending on how they interact with existing structural differences in access to care and digital infrastructure.(Campanozzi, 2023; Robinson, 2024; Tung et al., 2024)

To formalise this, we can represent each potential deployment site by a pair (U, R), where U is a vector of user-related attributes and R a vector of contextual resources:

$$U = (L, D, C), \qquad R = (H, N, G),$$

with L denoting language and health literacy, D digital literacy and familiarity with complex interfaces, and C clinical expertise and scope of practice; and H hardware and local device capabilities, N network and connectivity, and G governance structures (policies, training, support roles). We can then define a heuristic "readiness" function $\rho(U, R) \in [0, 1]$ describing how well SynapseCore is likely to fit the combination of user and context:

$$\rho(U, R) = f(L, D, C, H, N, G),$$

where f is high when language, digital skills, hardware, connectivity and governance all meet or exceed implicit thresholds, and low when one or more components fall short.(Campanozzi, 2023; Tracy, 2022) The present implementation has been implicitly tuned to contexts in which $\rho(U, R)$ is high for English-speaking psychiatrists working in well-resourced clinical settings; this creates important limitations for other user groups and environments.

17.3.1 Linguistic scope, localisation, and cultural framing

At present, the majority of SynapseCore's content, interface labels, flow names, and prompt templates are authored in English. Portions of the content library include Turkish phrasing or bilingual fragments (e.g. selected psychoeducational cards and risk phrasings), but there is no full-featured localisation layer that would guarantee systematic coverage of Turkish or other languages at the same level of detail as English. Upstream AI providers may offer

multilingual capabilities, but these are separate from the deterministic parts of the workbench and subject to their own limitations.(Torous et al., 2018; Tracy, 2022)

We can think of the linguistic support set as $\mathcal{L}_{\text{impl}} \subset \mathcal{L}$, where $\mathcal{L}$ is the set of clinically relevant languages and $\mathcal{L}_{\text{impl}}$ the subset for which SynapseCore provides:

1. complete UI labels and navigation;
2. validated scale wordings and scoring interpretations;
3. risk and capacity phrasing in line with local legal norms; and
4. at least one example library of AI prompt templates reviewed by native-speaking clinicians.

Currently, English sits at the core of $\mathcal{L}_{\text{impl}}$, with partial Turkish coverage for selected components and essentially no support for other languages. For users working primarily in Turkish, this leads to a hybrid experience: structured scales (e.g. PHQ-9, GAD-7) and some risk flows can be used alongside Turkish narrative sections, but many interface-level elements, tooltips, and AI configuration options remain in English.

This limited linguistic scope has several clinical and implementation consequences:

- **Risk of subtle misinterpretation.** Even when clinicians are bilingual, small mismatches between English terminology (e.g. “insight”, “thought form”, “capacity”) and local legal or clinical constructs can lead to misalignment in documentation and handover.(Tracy, 2022)
- **Barrier to non-English-speaking clinicians.** Clinicians whose primary working language is not English may find the interface cognitively burdensome, particularly under time pressure or when managing high-risk scenarios.
- **Lack of patient-facing localisation.** SynapseCore is primarily clinician-facing; it does not currently provide patient-facing materials (e.g. consent forms, psychoeducation handouts) in multiple languages, which can limit its utility in multilingual populations.(Robinson, 2024)

In addition to language per se, cultural framing of mental health concepts is only weakly modelled.(Campanozzi, 2023) The content library references diagnostic constructs and idioms of distress that align with mainstream, high-income settings. Adapting SynapseCore for use in other cultural contexts would require not only translation, but also co-designed reauthoring of examples, metaphors, psychoeducational materials, and risk formulations with local clinicians and service users.

17.3.2 Digital literacy and interactional complexity

The user interface of SynapseCore is intentionally rich: it employs a multi-pane layout, segmented controls, nested flow steps, and dynamic cards. This design is optimised for clinicians who are comfortable navigating complex EHR-like interfaces and who routinely work with structured data, filters, and timers. In effect, the current build assumes that users have at least a moderate level of digital literacy and prior exposure to electronic health records or comparable systems.(Campanozzi, 2023; Tracy, 2022)

Empirical studies in digital health consistently show that digital literacy is a key determinant of whether users can benefit from telemedicine and digital mental health tools.(Campanozzi, 2023; Kozelka, 2023) Individuals with low digital literacy—including some clinicians and many patients with serious mental illness—may be excluded from potential benefits, or may experience increased frustration and cognitive load. For SynapseCore, the assumed digital literacy manifests as expectations that users will:

- navigate between tabs and sections without losing track of context;
- understand conventions such as autosave indicators, tooltips, and modal dialogs;
- interpret status icons and coloured risk labels correctly; and
- manage browser tabs and windows while simultaneously interacting with the institutional EHR and other systems.

We can formalise this by introducing a scalar $D \in [0,1]$ representing a user's effective digital literacy in relation to SynapseCore's interface. The probability p_{eff} that the workbench will improve rather than worsen a user's cognitive burden can be conceptualised as an increasing function of D, with a threshold $D_{\min}$ below which complexity is more likely to hinder than help:

$$p_{\text{eff}}(D) = \sigma(\alpha(D - D_{\min})),$$

where σ is a logistic function and $\alpha > 0$ controls the sensitivity. The current design is implicitly calibrated for contexts where most users have $D \geq D_{\min}$; in settings where digital literacy is lower, the same interface may reduce usability and satisfaction, echoing challenges reported for EHR systems more broadly.(Holmgren, 2024; Schwappach, 2025)

Importantly, this limitation interacts with training and support structures. Emerging roles such as "digital navigators" in mental health services illustrate how dedicated personnel can bridge gaps in digital literacy for patients; analogous roles or targeted training may be needed for clinicians learning to use complex workbenches like SynapseCore.(Choudhary, 2025; Tracy, 2022)

17.3.3 Hardware, browser, and network constraints

As a browser-only application with a multi-pane layout, SynapseCore performs best on relatively modern hardware (e.g. contemporary laptops or desktops) with sufficient RAM and CPU capacity, and on browsers with robust support for contemporary JavaScript and graphics features. In practice, the reference environment during development has been a full-size display (or dual monitors), a recent version of a Chromium-based browser, and reliable broadband.

Let H denote an index of hardware capability (combining CPU, GPU, RAM, screen size), N an index of network quality (bandwidth, latency), and B a binary indicator for a supported browser family. We can define a simple viability condition for comfortable use:

$$\mathbb{I}_{\text{viable}}(H, N, B) = \mathbb{I}(H \geq H_{\min}) \cdot \mathbb{I}(N \geq N_{\min}) \cdot \mathbb{I}(B = 1),$$

where $\mathbb{I}$ is the indicator function. In many high-income settings, this condition will be satisfied; however, in resource-constrained environments or on older clinic hardware, one or more factors may fall below the implicit thresholds.(Perret et al., 2023; Tung et al., 2024)

Concretely, this leads to several limitations:

- **Limited support for small screens and mobile devices.** SynapseCore is not currently optimised for smartphones; attempting to use it on very small screens would likely result in cramped layouts and a poor user experience.
- **Sensitivity to low-spec or shared machines.** In clinics where clinicians share older workstations or where RAM/CPU resources are limited, performance may degrade, particularly when multiple tabs or large flows are open simultaneously.
- **Dependence on stable connectivity.** While deterministic flows and scoring functions work offline, AI-assisted features and some export pathways require network access. In

regions with intermittent connectivity or low bandwidth, these features may be unreliable, exacerbating the digital divide in access to advanced tools.(Robinson, 2024; Tung et al., 2024)

These constraints mirror broader concerns that digital health tools, if designed for high-resource settings, may inadvertently worsen health inequity by privileging organisations and clinicians who already have strong digital infrastructure.(Campanozzi, 2023; Tung et al., 2024)

17.3.4 Organisational readiness, governance, and support

User and context limitations are also shaped by organisational factors: policies, training programmes, supervision structures, and information governance. The SynapseCore design assumes that institutions deploying the workbench will provide:

- clear local policies on how AI-assisted text may be used and edited before entering the EHR;
- training sessions for clinicians on the boundaries of the tool (e.g. non-diagnostic, non-prescriptive);
- support channels for technical and workflow questions; and
- alignment with broader digital strategies (e.g. telepsychiatry, patient portals, patient-reported outcome measures).

However, in many services, digital mental health initiatives lack dedicated governance structures and may be deployed piecemeal, with variable oversight and resourcing.(Robinson, 2024; Tracy, 2022) In such environments, the same tool that enhances practice in one clinic may cause fragmentation or confusion in another.

Let G represent an organisational readiness index incorporating leadership support, policy clarity, training provision, and availability of local champions. The effective impact of SynapseCore in any given site can be viewed as a product of tool quality and organisational readiness:

$$\text{Impact} \approx \text{Tool\_quality} \times G.$$

Even if the intrinsic quality of SynapseCore is held constant, low G values (e.g. minimal training, unclear policy, no champions) will depress impact and may increase the risk of misinterpretation or inconsistent use.

17.3.5 Equity implications and the evolving digital divide

The intersection of language, digital literacy, hardware, and organisational readiness has important equity implications. The literature on digital mental health increasingly recognises that digital health literacy and access to infrastructure function as "super social determinants of health" that condition who can benefit from new technologies.(Campanozzi, 2023; Nutakor, 2024; Perret et al., 2023) People with serious mental illness, older adults, and socioeconomically marginalised groups often experience a compounded digital divide: reduced access to devices and broadband, lower digital literacy, and lower trust in digital systems.(Kozelka, 2023; Robinson, 2024)

SynapseCore presently offers limited built-in mechanisms to mitigate these structural inequities. Being clinician-facing, it can in principle help clinicians serve vulnerable patients more systematically; yet if only well-resourced clinics with digitally confident staff can deploy it effectively, there is a risk of widening gaps between services. In global mental health

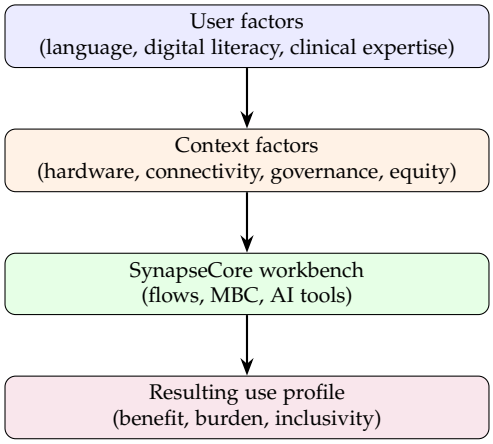


Figure 56: *Conceptual overview of user and context dependencies. The impact of SynapseCore depends not only on internal design but also on user factors (language, digital and clinical skills) and contextual resources (hardware, connectivity, governance). Misalignment in any layer can limit benefit or exacerbate inequities.*

contexts, where connectivity and multilingual, culturally adapted materials are particularly critical, the current English-centric, browser-only, high-spec assumptions are especially constraining.(Robinson, 2024; Torous et al., 2018)

In the longer term, addressing these user and context limitations will require a combination of:

- multilingual, culturally co-designed content and UI localisation;
- explicit integration of digital health literacy training for clinicians, potentially leveraging emerging "digital navigator" roles;
- intentional optimisation for lower-spec hardware and lower-bandwidth environments; and
- rigorous evaluation of how deployments affect equity metrics (e.g. service utilisation, outcomes, and satisfaction across demographic groups), rather than only aggregate averages.(Campanozzi, 2023; Tracy, 2022; Tung et al., 2024)

Until such steps are taken, SynapseCore should be regarded as a tool whose current sweet spot lies in relatively well-resourced, digitally mature services, with careful attention to the risk of reproducing or amplifying existing inequities.

Table 81: *Illustrative overview of user and context assumptions in the present SynapseCore build, associated risks when they are violated, and potential mitigation strategies.*

Dimension	Implicit assumption in current build	Risk if assumption is violated	Possible mitigation
Language and localisation	Clinicians can work comfortably in English; partial Turkish content is sufficient; patient materials managed outside SynapseCore	Misinterpretation of clinical concepts; reduced usability in non-English-dominant contexts; limited support for multilingual patients	Co-designed localisation; bilingual UI and content; integration of locally validated scale translations and risk phrasing
Digital literacy and interface familiarity	Users are accustomed to complex EHR-style interfaces and navigation patterns	Increased cognitive load; workarounds; underuse or misuse of advanced features by clinicians with lower digital literacy	Targeted training; progressive disclosure of complexity; collaboration with digital navigators; user testing in diverse clinician groups
Hardware and connectivity	Access to modern desktops/laptops, large screens, and stable broadband	Slow or unreliable performance; unusable layouts on small screens; uneven access to AI features across sites	Optimisation for lower-spec hardware; offline-friendly modules; clear documentation of minimum requirements; institutional investment in infrastructure
Organisational readiness and governance	Presence of policies, training, and champions to guide responsible use	Inconsistent practice; unclear accountability; unrealised benefits in settings without governance structures	Site-specific implementation plans; governance frameworks; integration with existing digital health strategies and supervision structures
Equity and inclusion	Early adopters are typically well-resourced services with strong digital infrastructure	Potential widening of gaps between services and populations; exclusion of those with low digital health literacy or weaker infrastructure	Equity-focused deployment strategies; monitoring of utilisation and outcomes across groups; prioritisation of support for under-resourced settings

18 Future Work

The present SynapseCore build should be understood as a deliberately scoped, browser-only workbench with strong internal structure but limited external connectivity. Future work therefore centres on opening carefully designed interfaces between the workbench and institutional ecosystems—most notably electronic health records (EHRs) and HL7 FHIR-based infrastructures—while retaining the privacy and observability principles set out in Sections 16.2 and 17.1. In parallel, subsequent subsections (not elaborated here) will address model lifecycle management for AI components, expanded clinical coverage, and evaluation programmes.

In this section we focus on the first and arguably most consequential strand of future work: robust, standards-based integration with EHR systems via FHIR and related interoperability frameworks.(Bender & Sartipi, 2013; HL7 International, 2019; Mandel et al., 2016) The aim is to transform SynapseCore from a self-contained workbench requiring manual copy/paste into a first-class, externally launchable SMART-on-FHIR application that can ingest structured

data, synchronise notes and MBC scales, and contribute to longitudinal patient records in a traceable manner.

18.1 EHR and FHIR Integration

Section 17.1 highlighted that the current SynapseCore implementation maintains no programmatic synchronisation between its internal state $S_t(i)$ and the institutional EHR record $R_t(i)$ for patient i at clinical time t. All transfers are human-mediated via exports and copy/paste. Future work aims to introduce an optional, standards-based integration layer that:

1. ingests structured data (problems, medications, observations, questionnaires, orders) from the EHR into the SynapseCore workbench;
2. supports bi-directional synchronisation of key artefacts such as notes, orders, and MBC scales; and
3. preserves strong privacy guarantees and clear provenance for all data flows.(Bender & Sartipi, 2013; HL7 International, 2019; Mandel et al., 2016)

We envision an architecture in which SynapseCore remains primarily a browser-side application, but is optionally paired with a small, institution-managed backend adapter that communicates with the EHR via HL7 FHIR and, where available, SMART-on-FHIR launch flows.(Dolin et al., 2016; Mandel et al., 2016) This adapter would be responsible for enforcing access control, mapping SynapseCore artefacts to FHIR resources, and managing synchronisation logic.

Formally, we extend the earlier decomposition $\mathcal{S} = \mathcal{U} \circ \mathcal{F} \circ \mathcal{A} \circ \mathcal{D}$ by introducing an external integration functor $\mathcal{I}$ that mediates between SynapseCore and the EHR:

$$\mathcal{S}_{\text{int}} = \mathcal{I} \circ \big(\mathcal{U} \circ \mathcal{F} \circ \mathcal{A} \circ \mathcal{D}\big),$$

where $\mathcal{I}$ encapsulates FHIR-based ingestion and synchronisation. In concrete terms, $\mathcal{I}$ is realised as a constrained set of RESTful operations over FHIR resources, with carefully defined mappings from and to SynapseCore internal state.

18.1.1 FHIR modelling of SynapseCore artefacts

The first step towards FHIR integration is to define a canonical mapping from SynapseCore artefacts—flows, scales, notes, risk formulations—to FHIR resource types. Let $\mathcal{A}_{\text{SC}}$ denote the set of SynapseCore artefacts and $\mathcal{F}$ the set of FHIR resources exposed by the institutional server. We seek a mapping

$$\mu : \mathcal{A}_{\text{SC}} \longrightarrow \mathcal{F}^{*},$$

which assigns to each artefact $a \in \mathcal{A}_{\text{SC}}$ a finite tuple of FHIR resources representing it in the EHR. For example, a single SynapseCore MBC scale administration may correspond to a `Questionnaire` definition, a `QuestionnaireResponse`, and one or more derived `Observation` resources.(Bender & Sartipi, 2013; HL7 International, 2019)

A preliminary mapping table is shown in Table 82, to be refined in collaboration with local informatics teams. The goal is to use FHIR resources that align with existing clinical data models and quality reporting pipelines, avoiding bespoke extensions where possible.

An important design choice is to ensure that resource creation is *transparent and reversible* from the clinician's perspective. For example, the clinician should be able to inspect, within

Table 82: *Illustrative mapping between SynapseCore artefacts and FHIR resources for future EHR integration. Actual mappings will need to be adapted to local profiling constraints and institutional implementation guides.*

SynapseCore artefact	Content	Candidate FHIR representation	Notes
MBC scale administration	PHQ-9, GAD-7, BFCRS items, scores and severity bands	`Questionnaire` (definition), `QuestionnaireResponse` (item-level data), derived `Observation` (total score, band)	Leverages existing PROM reporting patterns and quality metrics
Risk/safety assessment flow	Structured suicidality, self-harm, violence risk formulations	`Observation` with risk profile codes; `Condition` for persistent risk states; `CarePlan` or `ServiceRequest` for safety interventions	Requires careful profiling to reflect local medico-legal norms
SynapseCore encounter note	Clinician-edited narrative generated with optional AI assistance	`Composition` or `DocumentReference` linked to `Encounter` and `Patient`	Provenance metadata should capture level of AI involvement
Orders and referrals (future)	Requests for labs, imaging, psychology, social work follow-up	`ServiceRequest`, `MedicationRequest`, `Task`	Exact resource choice depends on local order entry practices
Structured symptom profiles and functioning	Time series of scores and functioning indices	`Observation` (panel or series) bound to standard codes (e.g. LOINC)	Enables reuse in quality improvement and research registries

the EHR, how a given SynapseCore risk formulation has been decomposed into one or more `Observation` and `CarePlan` resources, with clear provenance indicating its origin in SynapseCore and the degree of manual editing applied before finalisation.(HL7 International, 2019; Mandel et al., 2016)

At the level of data models, FHIR integration opens an opportunity to align SynapseCore content with established terminologies and value sets (e.g. ICD codes for diagnoses, SNOMED CT for clinical concepts, LOINC for lab and questionnaire items).(Bender & Sartipi, 2013; HL7 International, 2019) Future work will involve constructing mapping tables and profiles that bind SynapseCore artefacts to these standard vocabularies where clinically appropriate, while retaining the flexibility to incorporate local codes in institution-specific extensions.

18.1.2 Bi-directional synchronisation semantics

Beyond modelling artefacts, the central integrative task is to define bi-directional synchronisation semantics between SynapseCore and the EHR. Let $R_t(i)$ denote the EHR state for patient i at time t, and $S_t(i)$ the SynapseCore state as in Section 17.1. We aim to introduce mappings:

$$\Phi_t : R_t(i) \to S_t(i), \qquad \Psi_t : S_t(i) \to R_t(i),$$

representing ingestion into the workbench and commit back into the EHR, respectively. In practice, Φ_t will be realised as a pull of relevant FHIR resources (e.g. active problems, current medications, recent observations and QuestionnaireResponses), while Ψ_t will push newly created or updated resources (notes, scales, risk assessments) back to the EHR.(Boxwala et al., 2011; Mandel et al., 2016)

Two classes of synchronisation problems must be addressed:

1. *Staleness and conflict resolution.* EHR data may change between the time SynapseCore ingests it and the time the clinician attempts to commit updates. We must ensure that Ψ_t checks for conflicts (e.g. a note modified in the EHR since ingestion) and prompts the clinician to reconcile differences rather than silently overwriting.(Boxwala et al., 2011)
2. *Partial synchronisation and scope limitation.* For privacy and cognitive reasons, Φ_t should fetch only data that are relevant to the current encounter and task. This implies a scope function $\sigma : \text{Encounter} \to 2^{\mathcal{F}}$ that determines which FHIR resources are in scope for synchronisation, balancing completeness against overload and privacy.(Dolin et al., 2016)

A desirable property of the synchronisation protocol is that it be *monotonic* for certain classes of artefacts. For example, once an MBC scale has been committed as a `QuestionnaireResponse` and `Observation` pair, later edits to the SynapseCore view should not silently alter the committed FHIR resources; instead, a new assessment at time t' should generate new resources. Formally, for such artefacts the composition $\Psi_{t'} \circ \Phi_t$ should respect a versioning invariant:

$$\forall t' > t : \quad \text{version}\big(\Psi_{t'} \circ \Phi_t(a)\big) > \text{version}\big(\Psi_t(a)\big),$$

where $\text{version}(\cdot)$ is a monotonically increasing identifier (e.g. a timestamp or explicit version field).

From a user interface perspective, synchronisation must remain comprehensible to clinicians. We anticipate patterns such as:

- clear visual markers of which elements in SynapseCore are already committed to the EHR and which exist only locally;
- a small number of explicit synchronisation actions (e.g. a “Commit note to EHR” button) rather than hidden auto-sync; and
- an audit trail view summarising the FHIR resources created or updated during an encounter, aligned with institutional documentation norms.(American Psychiatric Association, 2021; Boxwala et al., 2011)

Technically, the FHIR adapter could implement an event-sourced pattern, where SynapseCore emits high-level events (e.g. “PHQ-9 completed”, “Risk formulation finalised”) and the adapter transforms these into one or more FHIR operations, ensuring that the set of emitted events is immutable and append-only. This design would facilitate downstream quality improvement and research analyses that rely on reconstructing sequences of clinical actions over time.

18.1.3 SMART-on-FHIR launch and contextualisation

A further strand of integration work involves enabling SynapseCore to be launched contextually from within the EHR using SMART-on-FHIR, where available.(Dolin et al., 2016; Mandel et al., 2016) In this pattern, the EHR launches an external app (SynapseCore) with a securely scoped access token and contextual parameters indicating the current patient, encounter, and user.

Let λ denote the SMART launch context, comprising:

$$\lambda = (\text{patient\_id}, \text{encounter\_id}, \text{user\_id}, \text{scope}, \text{launch\_token}).$$

The FHIR adapter would validate λ, derive the allowed operations and FHIR resource scope from scope, and initialise the SynapseCore session such that:

- the patient and encounter are pre-selected in the workbench;
- only FHIR resources permitted by the token are in scope for ingestion;
- any new artefacts created in SynapseCore are correctly linked back to the launching encounter and patient upon commit.

This approach offers several advantages:

- **Reduced context-switching.** Clinicians can move between the EHR and SynapseCore without re-entering identifiers or manually copying demographics.(Mandel et al., 2016)
- **Fine-grained access control.** The SMART token scope can restrict FHIR access to the minimum required for the current task, aligning with the privacy model described in Section 16.2.(HL7 International, 2019; U.S. Department of Health and Human Services, Office for Civil Rights, 2012)
- **Clear auditing.** EHR logs can record when SynapseCore was launched, for which patient, and with which capabilities, contributing to governance and post-market surveillance of digital tools.(American Psychiatric Association, 2021)

Future work will need to specify precise SMART scopes (e.g. read-only vs read-write, resource-level granularity) and to develop institution-specific implementation guides that describe how SynapseCore fits into existing application registries and consent models.

18.1.4 Clinical and research opportunities enabled by integration

Robust EHR and FHIR integration would not only reduce manual overhead but also open up new clinical and research possibilities. From a clinical perspective, tightly integrated MBC and risk/safety workflows could:

- improve adherence to screening and outcome measurement protocols by embedding PROMs directly into routine documentation;(Fortney et al., 2017a; Ko et al., 2023)
- enable real-time dashboards of symptom trajectories anchored in the authoritative EHR record rather than isolated local stores; and
- support institution-specific alerts and care pathways triggered by combinations of SynapseCore-derived observations and existing EHR data (e.g. suicidality flows combined with recent medication changes).

From a research and quality-improvement standpoint, FHIR-based integration would allow SynapseCore usage patterns and outputs to be incorporated into de-identified data warehouses and learning health system infrastructures, subject to appropriate governance.(American Psychiatric Association, 2021; Kilbourne et al., 2019) For instance, one could define a FHIR-based cohort of patients with repeated SynapseCore-supported MBC assessments and analyse:

- the association between structured risk formulations and subsequent adverse events;
- the impact of SynapseCore-assisted documentation on time-to-follow-up, treatment intensification, or adherence; and
- the differential uptake and benefit of SynapseCore across services and demographic groups.

Such work would require not only technical integration but also robust governance, including data access committees, patient and public involvement, and alignment with evolving regulatory frameworks for AI-enabled clinical decision support.(U.S. Food and Drug Administration, 2019; Vayena et al., 2018)

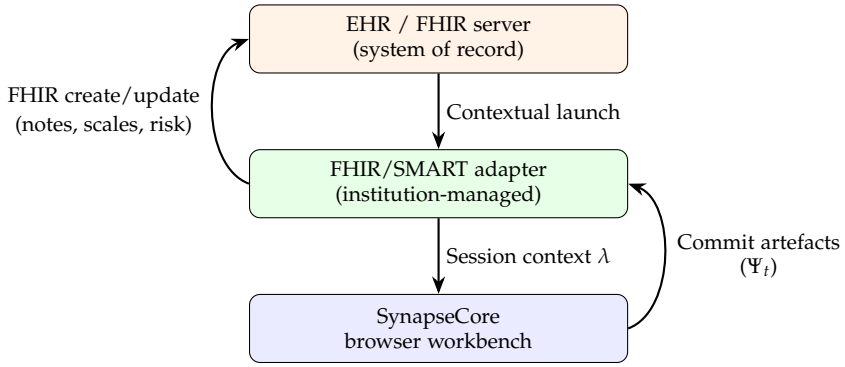


Figure 57: *Vertically organised architecture for future EHR and FHIR integration. An institution-managed adapter mediates between the SynapseCore browser workbench and the EHR, using SMART-on-FHIR for contextual launch and FHIR resources for curved, bi-directional synchronisation of notes, scales, and risk assessments.*

18.2 Advanced Analytics and Risk Models

SynapseCore already provides most of the primitives required for advanced analytics: a registry of structured encounters, a deterministic measurement-based care engine (Section 6), formalised risk grades via equations (6.14)–(6.15), and a session learning layer that treats timer traces as objects in a feature space (Section 14). The advanced analytics and risk models layer is conceived as a thin but powerful stratum on top of these primitives, designed to:

1. construct an explicit, auditable risk feature space from flows, MBC results, and contextual metadata;
2. learn probabilistic models for clinically important outcomes (e.g. crises, hospitalisation, persistent non-response) using Bayesian techniques that expose calibrated uncertainties; and
3. surface these models through explainable, longitudinal dashboards that help psychiatrists reason about trajectories and treatment response rather than single-point predictions, in line with the broader movement toward measurement-based care.(Belsher et al., 2019; Fortney et al., 2017a; Scott & Lewis, 2015)

Crucially, this layer is intended to remain *adjunctive* to clinical judgement and existing governance structures. SynapseCore does not promote a "black-box triage engine"; instead, it offers explicit feature maps, transparent probabilistic outputs, and SHAP-style explanations (Lundberg & Lee, 2017) that can be inspected, critiqued, and updated over time. This stance aligns with the conclusions of recent systematic reviews on self-harm and suicide prediction models, which emphasise calibration, interpretability, and careful integration into clinical workflows rather than headline discrimination metrics alone.(Belsher et al., 2019; Su et al., 2023)

18.2.1 From flows and MBC to a structured risk feature space

Let p index patients and $t \in \{1, \ldots, T_p\}$ index the encounters for patient p. For each encounter, Section 6 defined a vector of instrument totals $\mathbf{S}_{p,t}$, a composite MBC index $C_{p,t}$, and a discrete risk grade $G_{p,t} \in \{1, \ldots, 5\}$ via equation (6.15). The flows subsystem contributes structured outcomes $X_{p,t}^{(\text{flow})}$ such as suicidal ideation flags, self-harm history, access-to-means status, impulsivity markers, social support descriptors, and agreed observation or safety plans, all represented as well-typed JSON-like objects in the front-end.

Advanced analytics require these heterogeneous artefacts to be embedded into a stable

feature space. We therefore define a *risk feature map* $\phi_{\text{risk}} \colon \mathcal{X} \to \mathbb{R}^d$ of the form

$$\mathbf{x}_{p,t} = \phi_{\text{risk}}\big(\mathbf{S}_{p,t},\, C_{p,t},\, G_{p,t},\, X_{p,t}^{(\text{flow})},\, Z_{p,t}\big) \in \mathbb{R}^d, \tag{18.1}$$

where $Z_{p,t}$ denotes additional contextual covariates that can be collected without exposing raw narrative notes: for example, service line (general adult, old age, CAMHS, perinatal), referral source, coarse socio-demographic bands, recent non-attendance or crisis contacts, day-of-week and time-of-day, and simple treatment process indicators (e.g. "antidepressant recently started", "psychotherapy session number within current episode").

Following the design of the session ML feature map in Section 14.1, the risk feature map is explicitly structured into blocks:

- **Symptom and functioning block.** This block includes per-scale totals $S_{p,t}^{(\text{PHQ-9})}, S_{p,t}^{(\text{GAD-7})}, S_{p,t}^{(\text{PCL-5})}$, dichotomised severity bands, and change scores such as $\Delta S_{p,t}^{(\text{PHQ-9})} = S_{p,t}^{(\text{PHQ-9})} - S_{p,1}^{(\text{PHQ-9})}$, capturing both absolute burden and trajectory.
- **Risk grade and flags block.** The current grade $G_{p,t}$ and selected MBC-derived flags (e.g. "critical item endorsed", "severe side-effects") are encoded as one-hot or ordinal variables, reinforcing the semantics given to grades in equations (6.14) and (6.15).
- **Flow-derived risk block.** Structured outputs from safety, self-harm, capacity, substance use, and social context flows are mapped into a set of binary or ordinal indicators: presence of suicidal ideation, recent self-harm acts, access to lethal means, recent escalation of agitation, family or social support stability, housing instability, and so on. These are precisely the kinds of variables that clinical risk formulations already highlight, but here they are made explicit and machine-readable.
- **Treatment and process block.** This block encodes medication class and stage of treatment (initiation, titration, maintenance), psychotherapeutic modality, session count in the current episode, and appointment adherence history. Evidence from MBC implementation work suggests that combining symptom measures with process indicators improves the detection of non-response and deterioration.(Fortney et al., 2017a; Scott & Lewis, 2015)
- **Contextual block.** Coarse environment-level covariates such as clinic identifier, service configuration (e.g. home treatment vs. community clinic), and season of the year can be added where appropriate. These support hierarchical modelling without embedding detailed geospatial or socio-economic identifiers at the individual level.

By construction, ϕ_{risk} respects the privacy and minimisation principles set out in Section 16.2: no raw free text, no direct identifiers, and no fine-grained location information enter the feature space. Instead, the feature builder operates on already de-identified, structured elements that are available inside the browser, and exports only those fields that have been explicitly whitelisted for analytics in institutional governance documents.

18.2.2 Bayesian models for adverse event and trajectory risk

Given a feature vector $\mathbf{x}_{p,t}$ and an outcome of interest $Y_{p,t}$, the analytics layer treats risk modelling as a Bayesian inference problem. Following the literature on prediction models for self-harm and suicide attempts,(Belsher et al., 2019; Su et al., 2023) we start from a hierarchical logistic regression that balances flexibility with interpretability.

Let $Y_{p,t} \in \{0, 1\}$ indicate whether a pre-specified adverse event occurs within a fixed follow-up window after encounter t for patient p: for example, an unplanned psychiatric admission, a crisis team contact, or an emergency department presentation for self-harm

within 30 days. Conditionally on a latent risk $\pi_{p,t}$, we define

$$Y_{p,t} \mid \pi_{p,t} \sim \text{Bernoulli}(\pi_{p,t}), \tag{18.2}$$

$$\text{logit}(\pi_{p,t}) = \alpha_{s[p]} + \boldsymbol{\beta}^{\top} \mathbf{x}_{p,t}, \tag{18.3}$$

where $s[p]$ indexes the service or team responsible for patient p, $\alpha_{s[p]}$ is a service-level intercept capturing baseline risk, and $\boldsymbol{\beta} \in \mathbb{R}^d$ is a vector of regression coefficients over the feature space. Priors of the form

$$\alpha_s \sim \mathcal{N}(0, \sigma_\alpha^2), \qquad \beta_j \sim \mathcal{N}(0, \sigma_\beta^2) \tag{18.4}$$

encode conservative expectations about covariate effects and enable partial pooling across services. Hyperpriors on σ_α and σ_β can be chosen to reflect the desired degree of shrinkage.

Posterior inference is carried out in an *institution-managed analytics backend*, not in the browser, using a probabilistic programming framework such as Stan.(Carpenter et al., 2017) Given a training dataset $\mathcal{D} = \{(\mathbf{x}_{p,t}, Y_{p,t})\}$, the backend computes samples from

$$p(\alpha, \boldsymbol{\beta}, \sigma_\alpha, \sigma_\beta \mid \mathcal{D})$$

and exposes, via a narrow API, only:

- posterior predictive probabilities $\pi^*_{p,t} = \mathbb{E}[\pi_{p,t} \mid \mathcal{D}]$ or credible intervals for new encounters;
- summary statistics for $\boldsymbol{\beta}$ (e.g. posterior means and intervals) to guide model interpretation and governance;
- calibration curves and discrimination metrics at the cohort level; and
- diagnostic alerts if model performance drifts over time.

These probabilistic outputs can then be mapped back into the risk grade scale used in the MBC engine. Instead of viewing $G_{p,t}$ solely as a deterministic function of $C_{p,t}$, we introduce a probability-to-grade operator $\mathcal{G}_\theta$ such that

$$\widehat{G}_{p,t} = \mathcal{G}_\theta(\pi^*_{p,t}) = \begin{cases} 1, & 0 \le \pi^*_{p,t} < r_1, \\ 2, & r_1 \le \pi^*_{p,t} < r_2, \\ 3, & r_2 \le \pi^*_{p,t} < r_3, \\ 4, & r_3 \le \pi^*_{p,t} < r_4, \\ 5, & r_4 \le \pi^*_{p,t} \le 1, \end{cases} \tag{18.5}$$

with thresholds $0 < r_1 < r_2 < r_3 < r_4 < 1$ chosen via calibration and decision-analytic criteria. For example, thresholds can be tuned so that grade 5 corresponds to a risk level where the marginal benefit of intensified monitoring or crisis planning clearly outweighs its resource cost.(Belsher et al., 2019)

By treating $\pi^*_{p,t}$ as a quantity with a posterior distribution rather than as a point estimate, SynapseCore can expose uncertainty-aware visual encodings (e.g. shaded bands or error bars) in the UI, reinforcing the message that these are probabilistic risk assessments, not binary categorical judgements.

18.2.3 Explainability via SHAP-style decompositions

A recurring theme in the literature on clinical prediction models is that accuracy alone is insufficient; clinicians and governance bodies must be able to ask “why did the model consider this encounter high risk?” and receive a plausible, stable answer.(Lundberg & Lee, 2017) To

support this requirement, SynapseCore adopts SHAP (SHapley Additive exPlanations) as the default framework for explaining model outputs at the feature level.

Given a trained risk model $f : \mathbb{R}^d \to [0,1]$ and a feature vector $\mathbf{x}_{p,t}$, SHAP associates each feature $j \in \{1, \dots, d\}$ with an attribution $\phi_j(\mathbf{x}_{p,t})$ such that

$$f(\mathbf{x}_{p,t}) - \mathbb{E}_{\mathbf{X}}\big[f(\mathbf{X})\big] \approx \sum_{j=1}^{d} \phi_j(\mathbf{x}_{p,t}), \tag{18.6}$$

where the expectation is taken over a reference distribution of feature vectors.(Lundberg & Lee, 2017) The attributions ϕ_j can be interpreted as the contribution of feature j to moving the prediction away from the baseline risk given by the expectation.

In the SynapseCore design, SHAP values are computed in the analytics backend alongside the Bayesian model:

- *Local explanations* are generated on demand for a specific encounter (p,t), producing a small set of the largest positive and negative $\phi_j(\mathbf{x}_{p,t})$ values. These are transformed into a short explanation phrase in the encounter-level risk tile, for example: "Elevated PHQ-9 score, recent self-harm, and low social support contribute to higher risk; stable anxiety and no substance use partially offset risk."
- *Global explanations* are obtained by aggregating SHAP values over many encounters, yielding importance rankings and dependence plots that can be inspected by clinical leads and data scientists. Such plots can reveal whether the model is relying excessively on artefactual variables (e.g. clinic identifiers, documentation timings) or whether its behaviour aligns with consensus clinical knowledge.(Belsher et al., 2019; Su et al., 2023)

Because features in ϕ_{risk} are constructed from de-identified, structured elements, SHAP outputs can be shared in governance forums without exposing raw notes or identifiers. Furthermore, SHAP summaries can be stratified by protected attributes (e.g. age band, gender) to support fairness audits and to detect systematic differences in how risk is assigned across subgroups.

18.2.4 Longitudinal dashboards for trajectories and treatment response

Advanced analytics only become clinically useful when they are visualised in ways that align with how psychiatrists naturally reason about trajectories: symptom improvement or deterioration over time, the timing of interventions, and the occurrence of relapse or crisis events. The SynapseCore registry and MBC engine already provide the raw material for such reasoning; the analytics layer organises these into longitudinal dashboards.

For each patient p, we consider the time-indexed collection $\{(\mathbf{S}_{p,t}, C_{p,t}, G_{p,t}, \pi^*_{p,t})\}_{t=1}^{T_p}$, optionally augmented with event indicators $E_{p,t}$ (e.g. crisis contacts) and treatment markers $T_{p,t}$ (e.g. medication changes, therapy sessions). The dashboard presents this collection as a set of aligned time series:

- **Symptom and functioning trajectories.** Line plots of key scale totals and composite indices (PHQ-9, GAD-7, PCL-5, overall functioning), with clinically relevant thresholds shaded. Evidence from MBC trials suggests that such trajectories, when reviewed regularly, improve detection of non-response and guide timely treatment adjustments.(Fortney et al., 2017a; Guo et al., 2015; Scott & Lewis, 2015)
- **Risk grade and probability timelines.** A stepped plot of clinician-entered grades $G_{p,t}$ and model predicted grades $\widehat{G}_{p,t}$, overlaid with a smooth curve for $\pi^*_{p,t}$ and its credible band. Divergences between clinician and model can be explicitly marked, not as errors but as prompts for discussion and possibly model recalibration.

Layer	Mathematical object	Planned implementation hook
Risk feature construction	Feature map $\mathbf{x}_{p,t} = \phi_{\text{risk}}(\mathbf{S}_{p,t}, C_{p,t}, G_{p,t}, X_{p,t}^{(\text{flow})}, Z_{p,t})$ as in (18.1)	Front-end feature builder that serialises flow outcomes and registry snapshots into a typed `RiskFeatureVector`, operating only on de-identified, structured fields selected under the de-ID presets.
Bayesian risk model	Hierarchical logistic model with posterior $p(\alpha, \boldsymbol{\beta}, \sigma_\alpha, \sigma_\beta \mid \mathcal{D})$ and predictive probabilities $\pi_{p,t}^*$ as in (18.3)–(18.4)	Institution-managed analytics backend (e.g. Stan-based pipeline) that trains and validates models on governed exports, and exposes only calibrated risk estimates and diagnostics to SynapseCore.
Risk grade mapping	Probability-to-grade mapping $\widehat{G}_{p,t} = \mathcal{G}_\theta(\pi_{p,t}^*)$ with thresholds $r_1, \ldots, r_4$ as in (18.5)	Thin adapter that maps probabilities from the backend to the existing `RiskLevel` type and colour palette, plus calibration tools in cohort-level analytics views.
SHAP explainability	Local attributions $\{\phi_j(\mathbf{x}_{p,t})\}_{j=1}^d$ satisfying the additive decomposition in (18.6)	SHAP computation service co-located with the risk model, returning per-encounter and aggregated attributions for inspection and fairness auditing, without exposing raw text or identifiers.
Longitudinal dashboards	Trajectories $\{\mathbf{S}_{p,t}, C_{p,t}, G_{p,t}, \pi_{p,t}^*, E_{p,t}, T_{p,t}\}_{t=1}^{T_p}$	Registry- and episode-level views reusing the existing plotting infrastructure to render symptom trajectories, risk timelines, treatment markers, and event anchors for both individual patients and cohorts.

Table 83: *Planned advanced analytics stack on top of flows and the MBC engine. Each layer corresponds to a well-defined mathematical object and a concrete implementation hook, allowing institutions to adopt or disable components according to local governance while retaining a clear conceptual model of how risk estimates are produced.*

- **Treatment response markers.** Vertical bands or icons mark key treatment events (start or switch of antidepressants, introduction of mood stabilisers, initiation of psychotherapy), allowing clinicians to visually align changes in symptom burden and risk with interventions.
- **Event markers.** Crises, admissions, and emergency presentations (where available in the registry) are shown as discrete points, anchoring the narrative of the episode and providing concrete targets for retrospective case review and service-level learning.(Belsher et al., 2019)

At the point-of-care level, a compact "risk tile" in the encounter summary strip can condense this information: current grade $G_{p,t}$ with colour-coded styling from the existing risk palette, the latest modelled probability $\pi_{p,t}^*$ with a qualitative uncertainty indicator, a tiny spark-line of recent PHQ-9 or composite scores, and one or two SHAP-derived explanation bullets. At the cohort level, aggregated dashboards can present distributions and trends over time (e.g. proportion of patients above a given grade, rates of non-response), supporting quality improvement initiatives and model monitoring.

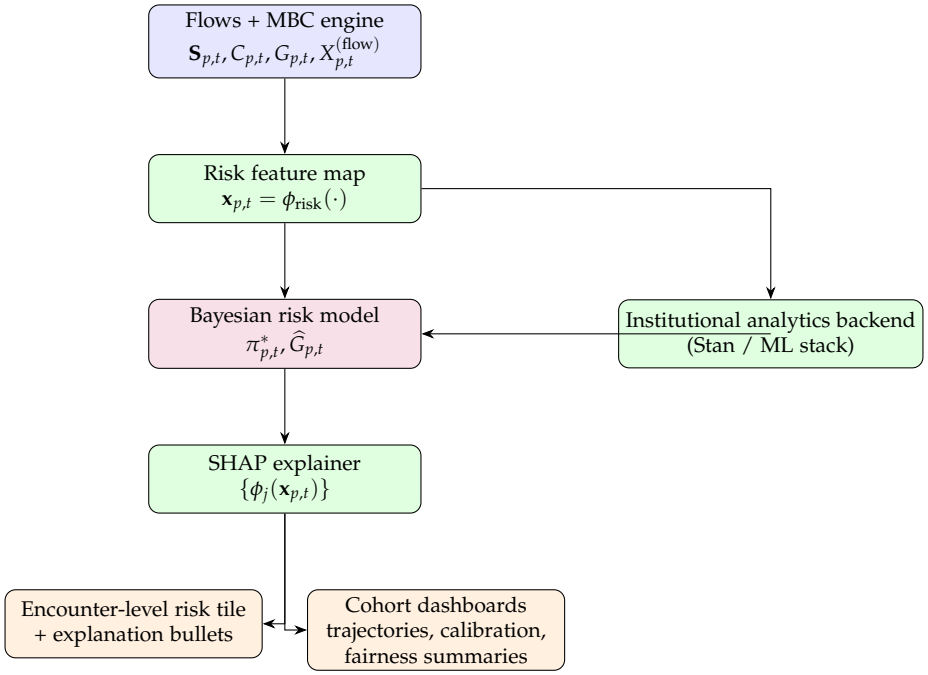


Figure 58: *Vertically organised pipeline for advanced analytics and risk models. Structured outputs from flows and the MBC engine are transformed by a risk feature map, sent as de-identified exports to an institutional analytics backend that maintains Bayesian risk models and SHAP explainers, and returned as calibrated risk estimates and feature attributions. These outputs feed both encounter-level risk tiles and cohort-level dashboards, while respecting the privacy and governance boundaries defined elsewhere in the SynapseCore architecture.*

18.3 Team-Based Workflows

A large proportion of psychiatric care is delivered not by isolated clinicians but by interdisciplinary teams that include psychiatrists, psychologists, nurses, social workers, occupational therapists, and trainees. Teamwork in such settings has been conceptualised as a dynamic process in which multiple professionals with complementary skills work towards shared clinical goals through interdependent collaboration, open communication, and joint decision-making.(Xyrichis & Ream, 2008) Collaborative care frameworks for depression, anxiety, and late-life mental disorders formalise this intuition: they couple measurement-based care (MBC), systematic case review, and stepped treatment intensification within a team-based, population-health perspective.(Richards et al., 2013; Unutzer et al., 2002) Similarly, early-intervention services for psychosis and youth mental health use multidisciplinary teams and structured case conferences to coordinate longitudinal care across inpatient and community settings.(McGorry et al., 2007; O'Connell et al., 2022)

At present, SynapseCore is deliberately scoped as a single-clinician, browser-resident workbench. The registry and flows framework encode patients, encounters, and structured flow runs as local data structures bound to the user's browser profile (Sections 6 and 7). Nevertheless, the underlying design anticipates future team-based extensions: the collaboration types in `src/types/collaboration.ts`, the `useCollaborationStore` state container (`src/store/useCollaborationStore.ts`), and the `CollaborationPanel` UI (`src/components/timer/CollaborationPanelPro.tsx`) already sketch a co-therapy and supervision layer that can be activated once a secure, institution-managed backend is available.

In this subsection we outline how SynapseCore can be extended from a single-user workbench to a team-aware environment supporting (i) real-time co-therapy, (ii) asynchronous supervision, and (iii) structured handover and case conferences, while preserving the pri-

vacy, observability, and medico-legal constraints articulated in previous sections. We proceed by introducing a conceptual model for teams and roles, mapping that model to the existing TypeScript types, and describing the supervisory and handover workflows that would sit on top of flows and MBC.

18.3.1 Conceptual model for teams, roles, and sessions

Let $\mathcal{U}$ denote the set of authenticated users (clinicians and trainees), $\mathcal{P}$ the set of patients, $\mathcal{E}$ the set of encounters, and $\mathcal{R}$ a finite set of roles. For team-based psychiatry, $\mathcal{R}$ typically includes roles such as $\{\text{therapist}, \text{supervisor}, \text{co-therapist}, \text{observer}, \text{admin}\}$, mirroring the `UserRole` union type in `src/types/collaboration.ts`.

In the current implementation, each encounter $e \in \mathcal{E}$ is implicitly associated with the local clinician using the workbench, and is represented by the `Encounter` interface in `src/centerpanel/registry/types.ts`. Future team-based versions introduce an explicit team-assignment relation $\mathcal{T}$ that maps each encounter to a finite set of $(\text{user}, \text{role})$ pairs:

$$\mathcal{T} : \mathcal{E} \to \mathcal{P}_{\text{fin}}(\mathcal{U} \times \mathcal{R}), \qquad \mathcal{T}(e) \subseteq \mathcal{U} \times \mathcal{R}. \tag{18.7}$$

Here $\mathcal{P}_{\text{fin}}(\cdot)$ denotes the finite power set. For a given encounter e, the set $\mathcal{T}(e)$ expresses both who is involved and in what capacity; for example,

$$\mathcal{T}(e) = \{(u_1, \text{therapist}), (u_2, \text{supervisor}), (u_3, \text{co-therapist})\}$$

would represent a supervised co-therapy encounter with one primary therapist, one supervisor, and one co-therapist.

Permissions and data visibility are then derived from role-specific policy functions. Let $\mathcal{A}$ be the set of allowed actions in the workbench (for example, {view_flows, edit_note, start_timer, export_summ
We define a role-to-permissions map

$$\Pi : \mathcal{R} \to \mathcal{P}_{\text{fin}}(\mathcal{A}), \tag{18.8}$$

where $\Pi(r)$ encodes which actions a user may perform when acting in role r. A user $u \in \mathcal{U}$ is authorised to perform action $a \in \mathcal{A}$ on encounter e if and only if there exists $(u, r) \in \mathcal{T}(e)$ such that $a \in \Pi(r)$. In practice, this map corresponds to the `TemplatePermissions` and co-therapy permission structures already sketched in the collaboration types, and would be enforced by the future backend adapter and front-end guards.

Sessions in SynapseCore are organised around the timer subsystem (Section 14), which already tracks laps, segments, and total session duration. For team workflows, we introduce a session-level event log that records the sequence of clinically relevant actions taken by each team member:

$$L_s = \big((t_i, u_i, a_i)\big)_{i=1}^{n}, \qquad t_1 \le t_2 \le \cdots \le t_n, \tag{18.9}$$

where s indexes a timer session, t_i is a timestamp, $u_i \in \mathcal{U}$ is the user performing the action, and $a_i \in \mathcal{A}$ is the abstracted action (for example, beginning a flow, inserting an MBC result, or appending a safety outcome paragraph). The log L_s provides a minimal sufficient statistic for reconstructing team-level activity during a session, and is the natural substrate for both supervisory review and medico-legal auditing.(Kilminster & Jolly, 2000)

Formally, a team-based session can then be represented as a tuple

$$S = \big(p, e, s, \mathcal{T}(e), L_s\big), \tag{18.10}$$

where $p \in \mathcal{P}$ is the patient, $e \in \mathcal{E}$ the active encounter, s the timer session identifier, $\mathcal{T}(e)$ the team assignment for that encounter, and L_s the ordered event log. In the current codebase, most of these fields are present in distributed form across the registry, timer state, and flow-run structures; the team extension consists of (i) making $\mathcal{T}(e)$ explicit, (ii) enriching the timer and flows with user identifiers, and (iii) persisting L_s in a backend-aligned format (for example, as part of a FHIR `Provenance` or `AuditEvent` resource).

18.3.2 Collaboration state and supervision workflows

The collaboration subsystem collects the entities needed for team workflows into dedicated types: Collaboration users (`CollaborationUser`), co-therapy sessions (`CoTherapySession`), workflow requests (`WorkflowRequest`), activity log entries (`ActivityLogEntry`), presence data (`UserPresence`), and template sharing metadata (`SharedTemplate`, `TemplatePermissions`). These types are centrally defined in `src/types/collaboration.ts` and orchestrated at runtime through `useCollaborationStore` in `src/store/useCollaborationStore.ts`.

From a clinical perspective, supervision and consultation workflows in psychiatry typically involve a trainee or primary clinician presenting a case, receiving structured feedback from a supervisor, and iterating on formulation and plan while maintaining clear accountability and patient safety.(Kilminster & Jolly, 2000) Evidence from collaborative care models underscores the value of regular case review conferences, in which registries of patients with elevated symptom scores are systematically reviewed, and treatment intensification is triggered when outcomes are not improving.(Unutzer et al., 2002; Whitebird et al., 2014) SynapseCore's flows and MBC dashboards already provide most of the structured information needed for such reviews; the collaboration layer is designed to bind that information into explicit workflows.

A `WorkflowRequest` instance can be understood as a typed supervision or consultation ticket:

$$w = \big(\text{id}, \text{type}, \text{status}, \text{requester}, \text{assignee}, \text{priority}, \text{payload}\big), \qquad (18.11)$$
$$\text{type} \in \{\text{supervision}, \text{consultation}, \text{peer\_review}, \text{approval}\},$$

where the payload references one or more encounters, flows, and timer sessions. Supervision requests (`type = 'supervision'`) bundle the following elements:

- a snapshot of the patient's MBC trajectory (Section 6.5);
- a set of completed flows and their autogenerated outcome paragraphs (for example, safety, capacity, observation, or agitation outcomes);
- selected portions of the narrative note; and
- an explicit supervision question or learning goal (for example, "Is the current level of observation justified given the risk profile?").

The `CollaborationPanel` UI then surfaces these workflows in a team context, providing views for pending requests, recent decisions, and activity streams. The existing implementation is deliberately local-only, using browser storage to persist collaboration state; a future FHIR-aligned backend would map workflow requests to institutional artefacts such as internal tickets or EHR-integrated supervision tasks.

18.3.3 Supervision, co-therapy, and governance flows

Clinical supervision is not simply an administrative exercise; it is a key determinant of patient safety, trainee development, and professional well-being.(Kilminster & Jolly, 2000) Psychiatric

Use case	Clinical aim	SynapseCore elements
Real-time co-therapy	Joint session with trainee, supervisor, and patient; shared view of symptoms, risk, and plan.	Timer-driven session shell; flows for assessment, risk, and observation; `CoTherapySession` in `useCollaborationStore`; presence data for cursors and active sections.
Asynchronous supervision	Supervisor reviews cases outside the live encounter, focusing on formulation, risk, and treatment adequacy.	`WorkflowRequest` instances with `type = 'supervision'`; MBC trajectories; completed flow runs; `CompletedRunReviewShell` read-only view of generated paragraphs.
Shift handover	Safe transfer of responsibility between on-call clinicians with minimal loss of context.	Registry view filtered by risk and status; structured safety and observation outcomes with explicit handoff narratives; export recipes in the tools surface for secure summaries.
Case conferences	Multidisciplinary discussion of complex cases, aligning perspectives and long-term plans.	Aggregated views over multiple encounters; longitudinal dashboards for symptoms and functioning; team-tagged notes and supervision workflows linked to specific decisions.

Table 84: *Illustrative team-based use cases and the corresponding SynapseCore building blocks that can support them once a collaboration backend is available.*

supervision must balance space for reflection and formulation with clear decisions about risk management, observation level, and treatment strategy. Team-based workflows in SynapseCore therefore need to encode both pedagogical and governance aspects.

At a high level, a supervised session in SynapseCore can be described by the following steps, expressed in terms of the team-session tuple S from Equation (18.10):

1. **Session set-up.** The trainee (or primary clinician) selects patient p, opens or creates encounter e, and starts a timer session s. A team assignment $\mathcal{T}(e)$ is initialised with at least one (therapist) and one (supervisor) entry.
2. **Data collection and flows.** During the encounter, the trainee completes psychometric instruments (MBC), runs relevant flows (for example, safety, capacity, observation), and records narrative notes. Each action produces a corresponding entry in the event log L_s (Equation 18.9) with the acting user u_i.
3. **Supervision request.** At or after the encounter, the trainee raises a `WorkflowRequest` with `type = 'supervision'`. The request references p, e, s, attaches key flows and MBC data, and includes a supervision question.
4. **Supervisory review.** The supervisor opens the workflow, reviews the MBC trends, flow outputs, and narrative, and records feedback as structured comments or supervision notes. In future backend-integrated deployments, these notes can be mapped to an EHR supervision template or internal task system.
5. **Decision and documentation.** The supervisor updates the workflow status (for example, approved, modifications requested, or follow-up needed), and—where appropriate—the handover or treatment plan. A summary paragraph is generated or edited, and inserted into the encounter note with explicit attribution (for example, "Supervisory addendum by Dr X").

Governance considerations are embedded at several levels:

- The collaboration store differentiates user roles and permissions, ensuring that trainees

cannot override supervisor decisions, and that observers have read-only access.
- Supervision notes and decisions are tied to specific encounters and timestamps, supporting medico-legal traceability.
- In multi-institution deployments, team identifiers and organisation identifiers in the collaboration types enable scoping of data to the appropriate clinical service or training programme.

These design principles are compatible with established collaborative care guidelines, which emphasise measurement-based caseload review, stepped care, and clearly defined roles for case managers and psychiatric consultants.(Richards et al., 2013; Unutzer et al., 2002) They also align with early-intervention models in which multidisciplinary teams share responsibility for high-risk youth, but retain clear supervisory lines and documentation standards.(McGorry et al., 2007; O'Connell et al., 2022)

18.3.4 Handover and case conferences

Failures of communication during handover are a leading contributor to adverse events in hospital medicine, including psychiatry. In high-acuity environments such as emergency departments, short-stay units, and inpatient wards, handovers occur multiple times per day, and must convey both static facts (diagnoses, medications, legal status) and dynamic judgements (risk formulation, observation level, triggers for concern). Case conferences and multidisciplinary meetings extend this logic over longer time horizons, using structured discussion to align the team around complex presentations and long-term goals.

SynapseCore already contains several ingredients for safer handover. The safety and observation flows encode explicit justifications for enhanced observation or containment, including a dedicated `handoffAndSupervision` field in the observation form state (`src/centerpanel/Flows/types/ObservationFormState.ts`). The corresponding outcome builder in `src/centerpanel/Flows/builders/observationOutcome.ts` generates a sentence such as: "Handoff / supervision described as ...", which is then appended to the encounter note. Similarly, the completed-run review shell (`src/centerpanel/Flows/shells/CompletedRunReviewShell.tsx`) presents the fully generated paragraph in a read-only, medico-legally framed view, making it explicit that the text supports clinical communication rather than functioning as an order set.

Team-based workflows extend these mechanisms in three directions:

1. **Shift handover dashboards.** Registry filters can be configured to surface patients with high risk scores, active safety flags, or recent agitation/observation flows. For each patient, the team-facing handover view aggregates (i) the latest MBC scores and trends, (ii) active risk and observation flags on the encounter, and (iii) the most recent safety and observation outcome paragraphs, including handoff narratives.
2. **Case-conference packets.** For scheduled multidisciplinary meetings, a "conference packet" can be generated using the tools surface (Section 9), bundling longitudinal symptom trajectories, key flows (safety, capacity, catatonia, lorazepam challenge), and supervision notes. This packet can be exported as a de-identified PDF or structured JSON, depending on local policy, and stored in the EHR via the FHIR adapter outlined in Section 18.1.
3. **Cross-cover summaries.** During nights or weekends, cross-covering clinicians often have limited familiarity with the patient. A concise, automatically maintained "cross-cover summary" can be constructed from (i) the highest-risk flows, (ii) legal status flags, and (iii) recent supervision decisions, providing a rapid yet structured view of why current observation levels or restrictions are in place.

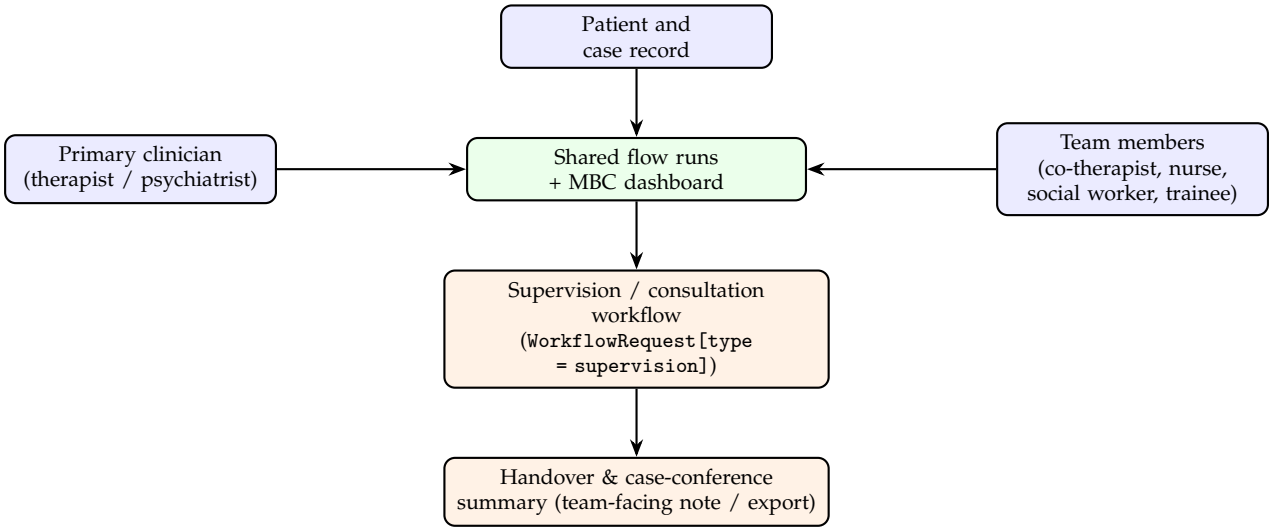


Figure 59: *Conceptual layout of team-based workflows in SynapseCore, from shared flow runs and MBC dashboards through supervision workflows to handover and case-conference summaries.*

In all three cases, the design goal is to improve the fidelity and efficiency of communication without replacing clinical judgement or altering local policies. Team-based views remain projections of the underlying patient, encounter, session, and workflow data, and any exports or synchronisation with external systems must respect the de-identification and consent constraints described in Section 16.2.

18.4 Domain Generalization

Although SynapseCore is instantiated here as a psychiatry-focused workbench, its underlying abstractions—registries of encounters, measurement-based care (MBC) calculators, structured flows, and AI-assisted documentation—are largely domain-agnostic. The same architectural pattern can support other specialties in which longitudinal symptoms, function, and risk must be tracked and acted upon: chronic pain services, neurology clinics, addiction programmes, and interdisciplinary liaison teams.(Kilbourne et al., 2018; Lewis et al., 2019) From a health-systems perspective, this type of cross-domain reuse is essential for managing multimorbidity, where psychiatric conditions, persistent pain, and neurological disorders frequently co-occur and interact to shape quality of life and service use.(Carlson & Yarns, 2023)

The present implementation already hints at such generalisation. The `SectionId` and `CardDoc` abstractions in `src/features/psychiatry/content/ContentSchema.ts` include explicit slots for "neuro-medical" content and follow-up monitoring; the `neuroMed` seeds in `src/features/psychiatry/seeds/neuroMed.ts` provide a delirium and encephalopathy workup scaffold that integrates safety and neurological reasoning. Similarly, the flows subsystem and MBC engine are parameterised over generic instrument definitions and outcome builders, rather than hard-coded psychiatric items. The question is therefore not *whether* SynapseCore can be used beyond psychiatry, but rather *how* to formalise and govern that generalisation.

In this subsection we sketch two complementary tracks:

1. *Clinical domain extensions*, where SynapseCore is adapted for pain medicine, neurology, and addiction services by swapping in domain-specific instruments, flows, and decision scaffolds.
2. *Educational simulators*, where the same abstractions are used to construct training en-

vironments and virtual OSCE-style stations for psychiatrists and other clinicians, with synthetic cases and formal scoring rubrics.(Issenberg et al., 2005)

We first develop a domain-general formalism, then illustrate how it can be instantiated in chronic pain, neurology, and addiction, and finally return to the educational use case.

18.4.1 Domain-agnostic core and configuration map

Let $\mathcal{D}$ denote the set of clinical domains of interest (e.g. adult psychiatry, chronic pain, movement disorders, addiction). For each domain $d \in \mathcal{D}$, we define a configuration tuple $\Theta^{(d)}$ that captures the domain-specific content and policies while leaving the core engine invariant:

$$\Theta^{(d)} = \big(\mathcal{S}^{(d)},\, \mathcal{M}^{(d)},\, \mathcal{F}^{(d)},\, \mathcal{R}^{(d)},\, \Pi^{(d)}\big). \tag{18.12}$$

Here:

- $\mathcal{S}^{(d)}$ is the set of section identifiers used to organise the left-rail navigation (for example, `"risk-safety"`, `"neuro-medical"`, or a hypothetical `"chronic-pain"` section);
- $\mathcal{M}^{(d)}$ is the set of measurement instruments and scoring rules (for psychiatry: PHQ-9, GAD-7, PCL-5; for pain: Brief Pain Inventory, numerical rating scales; for addiction: AUDIT-C, drug-screen instruments);(Bush et al., 1998; Cleeland & Ryan, 1994; Lewis et al., 2019)
- $\mathcal{F}^{(d)}$ is the library of structured flows (safety, agitation, observation, delirium workup, opioid risk assessment, and so on);
- $\mathcal{R}^{(d)}$ is the risk and response schema, defining severity bands, flags, and recommended actions for the domain;
- $\Pi^{(d)}$ is a domain-specific role-to-permissions map of the form in Equation (18.8), potentially constrained further by local regulation (e.g. addiction services under 42 C.F.R. Part 2 for substance use records).(U.S. Department of Health and Human Services, 2024)

The SynapseCore runtime can be viewed as a higher-order function $\mathcal{E}$ that takes a configuration $\Theta^{(d)}$ and a stream of user actions U_t and produces an evolving application state $S_t^{(d)}$:

$$S_{t+1}^{(d)} = \mathcal{E}\big(S_t^{(d)},\, U_t;\, \Theta^{(d)}\big), \qquad S_0^{(d)} \text{ given.} \tag{18.13}$$

Crucially, $\mathcal{E}$ is shared across domains: it encompasses the React/ TypeScript shell (`AppRoot`), registry and timer stores, flows host, MBC calculators, and tools surface. All domain-specificity is injected via $\Theta^{(d)}$, which is constructed from the content schema (`CardDoc`, `LocaleBlock`, `Reference` types), seed files (for example, `neuroMed.ts`), and domain-level flags in the configuration store (`usePsychiatryStore` and potential future `usePainStore`, `useAddictionStore`).

Formally, we can define a domain-generalisation map $\Gamma\colon \mathcal{D} \to \Theta$ that assigns to each domain $d \in \mathcal{D}$ a valid configuration $\Theta^{(d)}$ satisfying integrity constraints imposed by the engine (e.g. no orphan sections, all flows bound to known sections, all instruments mapped to scoring functions):

$$\Gamma(d) = \Theta^{(d)}, \qquad \Theta = \big\{\Theta' \text{ satisfying engine invariants}\big\}.$$

Domain adaptation then amounts to designing and maintaining Γ and its associated content pipelines, rather than modifying the core engine.

18.4.2 Chronic pain and pain medicine

Chronic pain clinics increasingly rely on structured patient-reported outcomes (PROs) and MBC principles to guide treatment, triage, and multidisciplinary care.(Dworkin et al., 2005; Turk & Melzack, 2011) Core outcome recommendations (IMMPACT) emphasise pain intensity, physical functioning, emotional functioning, and patient global impression of change as minimum measurement domains for clinical trials and, by extension, for measurement-informed routine care.(Dworkin et al., 2005) Brief, validated tools such as the Brief Pain Inventory (BPI) and numerical rating scales are widely used in daily practice.(Cleeland & Ryan, 1994)

SynapseCore's MBC engine already treats psychometric scales as generic objects with a scoring function σ and severity banding rules. For a pain service, we can define a pain-specific MBC configuration $\mathcal{M}^{(\text{pain})}$ with instruments such as BPI and PROMIS pain interference and map them into a composite index $C_{p,t}^{(\text{pain})}$ analogous to the psychiatric composite index in Section 6:

$$C_{p,t}^{(\text{pain})} = w_1 \, \widehat{I}_{p,t} + w_2 \, \widehat{F}_{p,t} + w_3 \, \widehat{E}_{p,t}, \qquad w_1 + w_2 + w_3 = 1, \tag{18.14}$$

where $\widehat{I}_{p,t}$ is a normalised pain-intensity score, $\widehat{F}_{p,t}$ a normalised functional-impairment score, and $\widehat{E}_{p,t}$ an emotional-functioning score at encounter t for patient p. Weights w_i can be tuned according to local priorities (e.g. emphasising function over intensity).

The flows library $\mathcal{F}^{(\text{pain})}$ for pain medicine might include:

- a *chronic pain intake* flow that captures pain location, quality, temporal pattern, red-flag features, and prior treatments;
- an *opioid risk and monitoring* flow, incorporating outcomes for opioid misuse risk, oversight plans, and patient agreements, integrated with AUDIT-C and other substance-use screens via existing MBC scaffolds;(Bush et al., 1998)
- a *non-pharmacological plan* flow that scaffolds physiotherapy, psychological therapies, and self-management strategies;
- flows for specific syndromes such as neuropathic pain, headache, or fibromyalgia, each with structured outcome paragraphs and safety flags.

In code, a pain-specific content package would mirror the psychiatry organisation:

- a `content` module defining new `SectionId` values (e.g. `"chronic-pain"`), domain-specific cards, and locale blocks with guidance and references;
- `seeds` files (for example, `painIntake.ts`, `opioidRisk.ts`) that define `CardSeed` objects with variables, prompt fragments, and hyperlinks to evidence-based guidelines;
- an MBC calculator module mapping raw BPI and PROMIS responses to scaled scores and severity bands, reusing the TypeScript patterns in `src/features/psychiatry/mbc/calculators.ts`.

Because pain medicine frequently intersects with mood, sleep, and substance use, a mixed-domain configuration is natural: the psychiatry instruments and risk flows can coexist with pain instruments within the same registry entry, with the domain flag in the registry (`registryDomain`) driving which left-rail sections and flows are highlighted for a given clinic or visit type.

18.4.3 Neurology and neuropsychiatric content

Neurology clinics manage a wide spectrum of conditions—epilepsy, movement disorders, demyelinating disease, stroke sequelae, cognitive impairment—that have strong neuropsy-

chiatric interfaces. Standardised outcome measures (for example, Expanded Disability Status Scale in multiple sclerosis, seizure-frequency diaries, cognitive screening tools) are increasingly embedded into routine care and clinical trials.(Kilbourne et al., 2018) The existing `neuroMed` seeds illustrate how SynapseCore can scaffold neurological reasoning within the same engine:

- the delirium/encephalopathy workup card provides a structured approach that is "CAM-ICU aware" while avoiding proprietary text, combines safety, neurology, and general medicine considerations, and generates a concise, medico-legally framed outcome paragraph;
- additional cards in the same seed group can cover acute focal deficits, seizure workup, and neuroprotective measures, each attached to the `"neuro-medical"` section in the left rail.

Formally, a neurology configuration $\Theta^{(\text{neuro})}$ would instantiate:

- $\mathcal{S}^{(\text{neuro})}$ with sections such as `"neuro-medical"`, `"stroke-followup"`, `"epilepsy"`, `"cognition"`, wired to domain-specific cards;
- $\mathcal{M}^{(\text{neuro})}$ with instruments and tracking fields for seizures, disability, cognitive function, and mood;
- $\mathcal{F}^{(\text{neuro})}$ with flows integrating safety, imaging, and neurologic examination into structured plans.

The advantage of using SynapseCore's generic flows host is that neurology flows can be expressed with the same primitives (field definitions, branching logic, legal copy, outcome builders) as psychiatric flows, while targeting a different evidence base and vocabulary. This is particularly appealing for liaison and consultation services, where psychiatrists and neurologists review the same patient from complementary angles and benefit from a shared, structured scaffold.

18.4.4 Addiction services and SUD-focused care

Addiction services have long used structured screening instruments and multi-domain assessments to characterise alcohol and drug use, related problems, and readiness to change.(Babor et al., 2007; McLellan et al., 1992) The Addiction Severity Index (ASI) introduced a multi-domain assessment framework spanning medical, employment, legal, family/social, and psychiatric domains, with composite scores that can be trended over time.(McLellan et al., 1992) Public-health approaches such as SBIRT (Screening, Brief Intervention, and Referral to Treatment) emphasise brief screening in general medical settings, motivational interviewing-based interventions, and clear referral pathways for high-severity cases.(Babor et al., 2007) AUDIT-C and related alcohol screens serve as brief initial instruments in this context.(Bush et al., 1998)

A SynapseCore configuration for addiction services, $\Theta^{(\text{SUD})}$, would adapt the existing abstractions as follows:

- **Instruments and composites.** $\mathcal{M}^{(\text{SUD})}$ would include AUDIT-C, drug-use screening tools, and domain-composite scores akin to ASI. The MBC engine would map these into composite indices $C_{p,t}^{(\text{SUD})}$ and risk bands reflecting overdose risk, withdrawal risk, and psychosocial instability.
- **Flows for SBIRT and care planning.** $\mathcal{F}^{(\text{SUD})}$ would contain flows for brief intervention (SBIRT-style), detoxification planning, maintenance treatment (e.g. opioid agonist therapy), and crisis/relapse planning. Outcome paragraphs would embed recommended follow-up intervals and supervision/handover elements aligned with regulatory norms for SUD programmes.(Babor et al., 2007; U.S. Department of Health and Human Ser-

Domain	Example instruments / PROs	Core flows and cards	Additional constraints
Chronic pain	Brief Pain Inventory (BPI), numerical pain ratings, PROMIS pain interference.(Cleeland & Ryan, 1994; Dworkin et al., 2005)	Chronic pain intake, opioid risk and monitoring, non-pharmacological plan, syndrome-specific scaffolds (neuropathic pain, headache).	Balance on pain intensity vs function; opioid stewardship; integration with mental-health instruments.
Neurology / neuropsychiatry	Seizure-frequency logs, disability scales, cognitive screening, mood instruments.	Delirium and encephalopathy workup, stroke and acute neurology flows, epilepsy and cognition follow-up plans.	Coordination between neurology, psychiatry, and general medicine; imaging and investigation pathways.
Addiction services	AUDIT-C, drug-use screens, ASI-style domains and composite scores.(Babor et al., 2007; Bush et al., 1998; McLellan et al., 1992)	SBIRT flows, detoxification and withdrawal plans, maintenance treatment scaffolds, relapse-prevention plans.	Enhanced privacy and consent requirements (e.g. 42 C.F.R. Part 2), stigma-sensitive documentation.

Table 85: *Illustrative domain configurations for pain medicine, neurology, and addiction services. Each domain instantiates SynapseCore's abstract configuration tuple $\Theta^{(d)}$ with domain-specific instruments, flows, risk schemas, and governance constraints, while reusing the same registry, MBC engine, and flows host.*

vices, 2024)

- **Privacy-aware exports.** Because SUD records often receive special legal protection (e.g. 42 C.F.R. Part 2 in the US), $\Pi^{(\mathrm{SUD})}$ and the tools surface (for exports) would enforce stricter policies on which events and narrative sections can be shared between teams and systems, in line with Section 16.2.

From an implementation standpoint, SUD flows and cards would live alongside psychiatric flows in the same feature package, with domain flags on patients or encounters determining which flows are promoted for a given clinical context (e.g. addiction clinic vs inpatient ward). The AI-assisted documentation layer could then provide domain-aware templates for motivational interviewing summaries, relapse-prevention plans, and medication-assisted treatment notes, while preserving the traceability and de-identification guarantees established elsewhere in the architecture.

18.4.5 Educational simulators and training environments

Beyond direct clinical deployment, the same abstractions can be used to build educational simulators for psychiatry and related fields. Simulation-based medical education has a substantial evidence base: systematic reviews highlight that high-fidelity simulations, when aligned with explicit learning objectives and accompanied by debriefing, can improve knowledge, skills, and behaviour in clinical trainees.(Issenberg et al., 2005) In psychiatry and mental-health training, simulated patients (standardised actors, virtual patients, and computer-based scenarios) have been used to teach diagnostic interviewing, risk assessment, and communication skills.(Matsumura et al., 2018)

SynapseCore's architecture is well-suited for such simulators because it already models:

- *structured information gathering* (flows, psychometric scales);

- *reasoning and formulation* (MBC indices, risk and treatment scaffolds);
- *documentation and communication* (structured notes, safety and handover narratives).

To turn these into a training environment, we introduce a *scenario generator* $\mathcal{G}$ that samples synthetic but clinically realistic cases:

$$H \sim p_\psi(H), \qquad O = \mathcal{O}(H), \qquad Y = \mathcal{Y}(H), \tag{18.15}$$

where H is a latent case description (history, mental state, co-morbid conditions, social context), O is the observable data stream from the trainee's point of view (answers to history questions, scale responses), and Y is a set of reference labels (e.g. diagnoses, risk formulation, "gold-standard" plan fragments) against which trainee performance is evaluated.

In practice:

- scenarios are encoded as extended `CardDoc` objects with additional fields for ground-truth labels and scoring rubrics;
- the flows host presents the trainee with history-taking and examination prompts, while the timer and registry record sequence and timing of actions;
- a scoring engine compares the trainee's extracted data and formulated plans with the reference labels Y, producing analytic feedback (e.g. missed risk factors, incomplete safety plan, over- or under-estimation of observation needs).

We can formalise the scoring as a mapping from the session event log L_s (Equation (18.9)) to a vector of competency scores $\mathbf{q} \in [0,1]^k$:

$$\mathbf{q} = \Phi_{\text{train}}\big(L_s, Y\big), \tag{18.16}$$

where components of $\mathbf{q}$ correspond, for example, to data-gathering quality, risk-recognition accuracy, appropriateness of the treatment plan, and quality of documentation. In a multi-domain simulator, Y may include pain-specific targets (e.g. adequate assessment of pain interference), neurological red flags, and addiction-specific SBIRT tasks, depending on the scenario.

Educational deployments raise additional governance questions. Trainee-facing instances would typically run on institution-controlled hardware with synthetic or heavily de-identified cases, and training data would be stored separately from clinical data. The privacy model in Section 16.2 still applies, but with a different emphasis: protecting trainee performance data, ensuring fairness in assessment (e.g. avoiding biased AI grading), and clearly separating formative simulation results from summative evaluations.

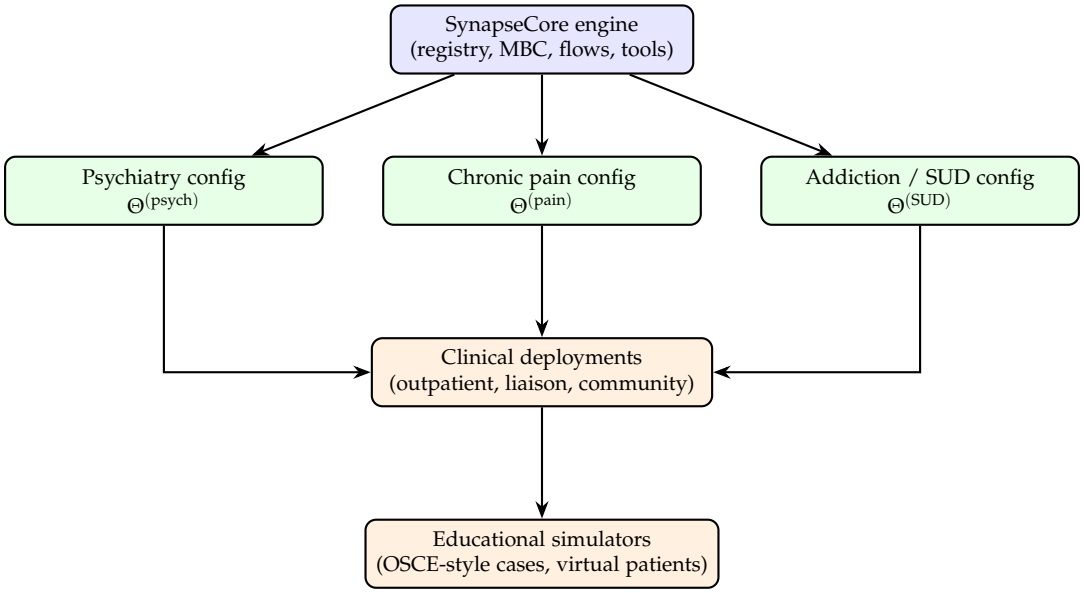


Figure 60: *Domain generalisation view of SynapseCore. A single engine is configured by domain-specific tuples $\Theta^{(d)}$ for psychiatry, chronic pain, addiction, and other domains. These configurations feed both clinical deployments and educational simulators, allowing content and governance to vary by domain while the core architecture remains stable.*

19 Conclusion

The SynapseCore workbench was designed from the outset as a concrete, inspectable answer to a practical question: what would it mean to embed large language models and measurement-based care (MBC) in everyday psychiatry without sacrificing clinical judgement, patient safety, or data protection?(Fortney et al., 2017a; Guo et al., 2015; Torous et al., 2018) Rather than proposing a monolithic "AI system" that operates behind opaque APIs, SynapseCore treats AI as a set of programmable, auditable components that sit inside a clearly specified architecture: a multi-model orchestration engine (Part 4), a psychiatry knowledge framework with a typed content schema (Section 5), a first-class MBC engine (Section 6), structured clinical flows and a session timer (Sections 7 and 14), and a tools and IDE layer for consultative work and local content engineering (Sections 9 and 11).

Across the manuscript we have argued for three complementary design principles. First, the system must be *evidence anchored*: structured flows, psychometric calculators, and AI prompts are all grounded in explicit, versioned artefacts, with clear links to existing guidelines and empirical literatures.(Fortney et al., 2017a; Unutzer et al., 2002) Second, it must be *clinician centred*: flows, timer behaviours, and AI-assisted notes are shaped around actual clinical tasks (safety assessment, observation decisions, handover, supervision), with deterministic fallbacks and conservative defaults. Third, it must be *privacy preserving and governable*: the reference implementation confines sensitive data to the browser, exposes de-identified export recipes instead of raw database access, and aligns its concepts with established privacy frameworks such as HIPAA and GDPR, allowing future institutional deployments to attach formal policies and monitoring to the same abstractions.(European Parliament and Council of the European Union, 2016b; U.S. Department of Health and Human Services, 2024; U.S. Department of Health and Human Services, Office for Civil Rights, 2012)

The remainder of this conclusion section consolidates these strands. Subsection 19.1 offers a synthetic summary of SynapseCore's architecture and capabilities, weaving together the AI orchestration engine, psychiatry content framework, MBC engine, flows and timer, and

tools/IDE stack into a single, multi-layered map. Subsequent subsections (not developed here) can elaborate on broader implications for digital psychiatry, training, and institutional adoption.

19.1 Summary of SynapseCore's Architecture and Capabilities

High-level architecture. From a software perspective, SynapseCore can be regarded as a composition of five interacting layers:

$$\mathcal{W} = \mathcal{I} \circ \mathcal{T} \circ \mathcal{F} \circ \mathcal{M} \circ \mathcal{A}, \tag{19.1}$$

where $\mathcal{A}$ is the multi-model AI orchestration engine (Section 4), $\mathcal{M}$ the MBC engine (Section 6), $\mathcal{F}$ the structured flows framework (Section 7), $\mathcal{T}$ the session timer and session-ML layer (Section 14), and $\mathcal{I}$ the combined psychiatry knowledge framework and clinical tools/IDE interface (Sections 5, 9 and 11). The operator "$\circ$" denotes functional composition over shared, typed state; in implementation terms, the layers are formalised as Zustand stores, typed configuration objects, and React components with well-annotated props.

Concretely, a typical workday for a psychiatrist using SynapseCore is structured as follows. The registry layer keeps track of patients and encounters, exposing typed handles that determine which flows, instruments, and tools are available at any given time. The psychiatry knowledge framework provides the left-rail taxonomy of sections (risk/safety, observation, capacity, psychotherapy, neuro-medical, follow-up, and so on) and attaches to each section a hierarchy of cards, flows, and reference material encoded as `CardDoc` trees. The session timer organises real-time work into segments and laps, emitting an event trace that can be consumed by the session-ML modules for pattern analysis (Section 14). The MBC engine processes psychometric instruments into deterministic scores, severity bands, and composite indices that can be consulted interactively or trended longitudinally. Flows then combine these elements—registry, timer, MBC outputs, and content—into explicit state machines whose accepting states generate legible paragraphs for notes, referrals, and handover.

Multi-model AI orchestration. The AI layer is designed to be provider-neutral and observable rather than hard-wired to a specific vendor. At runtime, `useAiConfigStore` maintains a normalised configuration of active providers and models, including sampling hyperparameters, credential bundles, and capability flags. The `samplingMapper` translates high-level sampling requests from the UI (for example, "risk-aware note drafting" or "consultation copilot") into provider-specific request objects, while preserving a shared schema for temperature, maximum tokens, and streaming behaviour. A small set of orchestration combinators—for example, parallel multi-model queries with reconciliation, or fallback chains between local and remote models—is expressed as pure functions over these configuration objects, allowing the same orchestration logic to be tested in isolation and then bound to real providers.

From a clinical perspective, this means that AI is never an all-or-nothing black box. Each AI-assisted affordance in the UI (such as an "explain this PHQ-9 trajectory" button, or a "draft safety section" helper) is anchored by a specific configuration, prompt template, and scope of authority described elsewhere in the manuscript. Because AI calls are wrapped in OpenTelemetry-compatible spans and enriched with route metadata, institutions can instrument real-world usage—measuring which models are called, with which parameters, and for which flows—without seeing raw clinical text.(European Parliament and Council of the European Union, 2016b; Torous et al., 2018) This separation enables future work on governance, differential privacy, and model comparison using de-identified traces.(Huckvale et al., 2019)

Psychiatry knowledge framework. At the content layer, SynapseCore moves away from free-form templates toward a typed, navigable knowledge framework. Sections are declared as a

closed union type (Equation 5.1), and each section hosts a tree of `CardDoc` objects that combine overview text, "quick use" guidance, structured variables, and cross-references. These trees are versioned and compiled into a registry that can be rendered in different UI shells (left rail, card detail views, flows documentation) and edited via the Enhanced IDE. The result is a psychiatry-specific knowledge base that is both human-readable and machine-tractable, suitable for direct consumption by flows, MBC calculators, and AI prompts.

Clinically, this translates into a more stable and transparent scaffolding for core psychiatric tasks. Risk and safety sections gather flows and cards related to suicide, violence, and observation; psychotherapy sections host content for modalities and inter-session planning; follow-up sections focus on review and long-term monitoring; neuro-medical sections bridge psychiatry, neurology, and general medicine, as discussed in Section 18.4. Because all of these are typed and stored in a single codebase, they can be safely extended over time, with institutional teams authoring their own content packs and applying them via the IDE in a controlled, auditable fashion.(Kilbourne et al., 2018; Lewis et al., 2019)

Measurement-based care engine. The MBC subsystem takes seriously the requirement that every numeric score be traceable, reproducible, and interpretable. Each instrument is defined by (i) a schema for item responses, (ii) a deterministic scoring function, and (iii) a mapping from raw scores to severity bands and composite indices. These are implemented as pure TypeScript functions with comprehensive unit tests, mirrored by transparent HTML autoscore blocks that make the scoring rules visible at the point of care. Composite indices, such as the general psychiatric composite $C_{p,t}$ or the pain-specific composite $C_{p,t}^{(\text{pain})}$ (Equation 18.14), are then derived as weighted combinations of normalised subscales, and mapped to risk grades $G_{p,t} \in \{1, \ldots, 5\}$ that structure dashboards and flows.

In practice, this enables a style of MBC that is both rigorous and usable. Instrument selection can be tuned to the setting (acute ward, outpatient, liaison), while the underlying engine guarantees consistency across encounters and clinicians.(Fortney et al., 2017a; Guo et al., 2015) Longitudinal dashboards, such as those used for exploratory analytics and risk modelling, are built directly on these well-typed objects rather than on ad hoc exports. Because MBC outputs are first-class citizens in the state model, they can drive flow branching, AI prompt conditioning, and export recipes without duplicating logic.

Structured flows and session timer. Structured clinical flows operationalise the idea that many high-risk or complex tasks—for example, suicide risk assessment, capacity evaluations, or agitation management—can be represented as finite directed acyclic graphs (DAGs) whose nodes correspond to well-defined intermediate states and whose edges encode safe transitions. SynapseCore formalises these flows as typed state machines, with clear entry, intermediate, and accepting states, and with associated outcome builders that generate well-structured paragraphs for clinical notes.

The session timer, implemented as a pure state machine, provides the temporal context for these flows. By segmenting encounters into laps and clinical segments, and by emitting an event log L_s, the timer layer turns the otherwise ephemeral rhythm of the clinical day into analysable data. Flows can attach themselves to timer segments (for example, linking a safety assessment to a particular 25-minute block), and future session-ML modules can use these traces to model workload, supervision needs, and patterns of care. Crucially, the same primitives support both single-clinician and team-based workflows, with co-therapy and supervision states layered on top of the base timer engine.

Tools surface and Enhanced IDE. The tools surface and Enhanced IDE, described in Sections 9 and 11, complete the picture by turning SynapseCore into a programmable consultation and development environment. On the clinical side, tools include calculators, export recipes, de-ID presets, and longitudinal report builders that allow psychiatrists to move flu-

Layer	Primary responsibilities	Key properties in SynapseCore
AI orchestration engine	Provider-agnostic configuration of language models; sampling and routing; telemetry for AI calls.	Normalised runtime configuration (`useAiConfigStore`); sampling mapper with pure combinators; OpenTelemetry spans for observability.(Torous et al., 2018)
MBC engine	Deterministic scoring of psychometric instruments; severity banding; composite indices and risk grades.	Pure scoring functions with mirrored HTML autoscore blocks; composite indices $C_{p,t}$ and $G_{p,t}$; longitudinal dashboards for outcome tracking.(Fortney et al., 2017a; Guo et al., 2015)
Flows and timer	Representation of high-stakes tasks as DAGs; session timing and segmentation; event logs for session ML.	Typed flow shells with accepting states and outcome builders; pure timer engine; session logs L_s for analytics and supervision support.
Psychiatry knowledge framework	Domain taxonomy and content; cards, sections, and reference material for core psychiatric tasks.	Typed section IDs and `CardDoc` trees; IDE-editable content packs; shared scaffolding for flows, MBC, and AI prompts.(Kilbourne et al., 2018; Lewis et al., 2019)
Tools surface and IDE	Clinical calculators, export and de-ID tools; local content engineering and AI-assisted authoring.	Tools tab with schema-backed recipes; Enhanced IDE with type-safe content editing, telemetry hooks, and AI co-pilots for knowledge maintenance.(Issenberg et al., 2005)

Table 86: *Summary of SynapseCore's principal architectural layers and their roles in everyday clinical and development workflows. Each layer is implemented as a type-safe, browser-resident module that can be configured, observed, and extended without compromising the privacy and traceability guarantees described elsewhere in the manuscript.*

idly between day-to-day care and population-level review. On the informatics side, the IDE hosts content packs, schema-aware editors, and AI co-pilots for drafting and refactoring cards, flows, and explanatory text.

This dual role—clinical workstation and local IDE—is essential for bridging the gap between generic AI models and the specific needs of psychiatric services. Institutional teams can iteratively adapt flows, notes, and content packs, while observability hooks provide feedback on usage patterns and potential failure modes. Because all of this sits within the same type-safe, browser-resident codebase, the risk of configuration drift between "sandboxes" and production is reduced, and local governance structures can be attached directly to the artefacts that clinicians actually use.

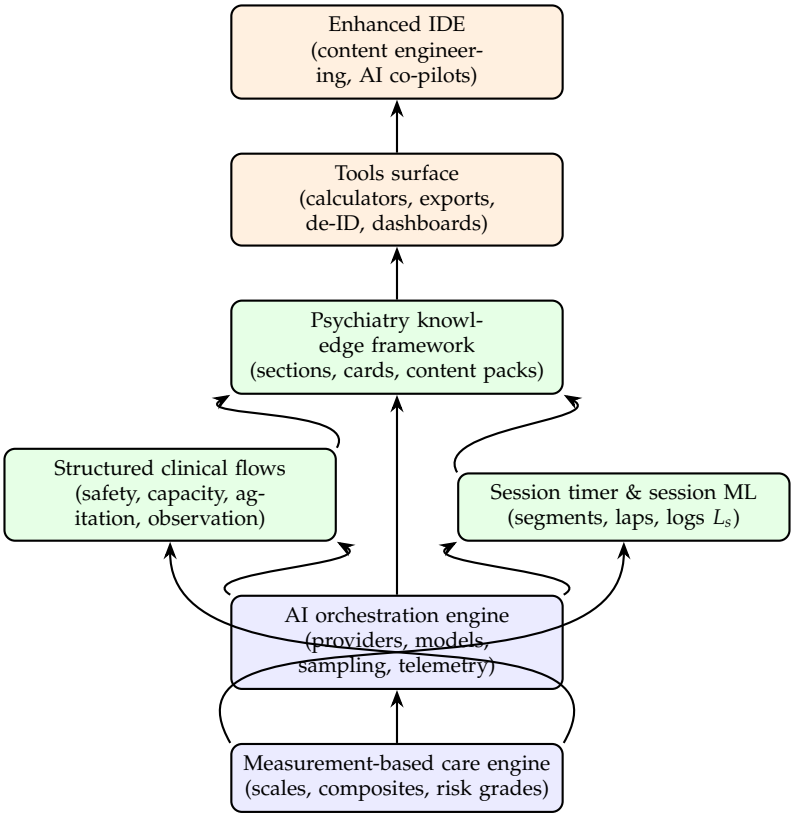


Figure 61: *Compact vertical summary of the main SynapseCore subsystems. The AI orchestration and MBC engines form the computational base; above them, flows, timer, and the psychiatry knowledge framework structure day-to-day clinical work; at the top, the tools surface and Enhanced IDE support consultative workflows and local content engineering.*

19.2 Implications for Digital Psychiatry and AI-Augmented Practice

SynapseCore sits within a broader movement in digital psychiatry that seeks to reconcile three tensions: (i) the growing documentation and administrative burden on clinicians, (ii) the demand for more structured, measurable and accountable care, and (iii) the rapid evolution of AI models and digital tools, often outpacing formal evaluation and governance.(Shanafelt et al., 2015; Torous et al., 2018) Rather than framing AI as a monolithic replacement for clinical judgement, SynapseCore presents a concrete pattern for *AI-augmented practice*: large language models (LLMs) and other components are embedded into well-specified workflows, backed by deterministic measurement-based care (MBC), and constrained by explicit privacy and observability guarantees (Sections 16.2 and 17).

In this subsection we highlight three implications of this design for digital psychiatry and related fields: (i) the potential to reduce documentation burden without erasing the clinician's narrative voice, (ii) the normalisation of structured, measurable care processes through MBC and flows, and (iii) the emergence of SynapseCore as a platform for experiment and rapid iteration in clinical services and training.

19.2.1 Reduced documentation burden without loss of nuance

Documentation burden has been repeatedly identified as a major driver of burnout in medicine, with psychiatrists and other clinicians often spending as much or more time on electronic health records (EHRs) as in direct face-to-face contact.(Arndt et al., 2017; Shanafelt

et al., 2015) In many services, the workday can be informally decomposed into

$$T_{\text{total}} = T_{\text{direct}} + T_{\text{doc}} + T_{\text{coord}},$$

where T_{direct} denotes direct clinical contact, T_{doc} time spent on documentation (notes, forms, letters), and T_{coord} time on coordination and communication. A crude but informative summary of documentation burden is the *burden ratio*

$$R_{\text{doc}} = \frac{T_{\text{doc}}}{T_{\text{direct}} + T_{\text{doc}}}. \tag{19.2}$$

Empirical estimates from EHR log studies suggest that R_{doc} may approach or exceed 0.5 in some ambulatory settings, meaning that documentation consumes at least as much time as direct care.(Arndt et al., 2017; Downing et al., 2018)

SynapseCore does not directly manipulate EHR billing fields or mandated forms; instead, it offers a layered strategy for reducing the *effective* T_{doc} required to achieve a given standard of care and communication:

- **Upstream structuring.** Flows and MBC instruments capture key facts, risk factors, and justifications in structured form during the encounter, rather than leaving them to be reconstructed ex post in free-text notes. Safety, observation, capacity, and agitation flows, for example, encode decision rationales that are then re-used by outcome builders for documentation (Sections 7 and 14).
- **Deterministic note skeletons.** Outcome builders generate legally salient paragraphs that cover minima for documentation (for example, explicit statements of observation level, risk formulation, and handover), which can be accepted, edited, or discarded by the clinician. This reduces the need to reconstruct complex reasoning under time pressure while preserving the ability to override or nuance the generated text.
- **AI-assisted refinement.** The multi-model AI layer is used to refine, summarise, or adapt drafts to different audiences (e.g. a handover vs. a patient-facing letter), under explicit prompts and within the constraints of the de-identification and logging model.(Nori et al., 2023; Singhal et al., 2023) AI is not asked to originate clinical content from scratch; it operates on structured inputs and drafts, maintaining traceability.

Put differently, SynapseCore aims to compress documentation effort by shifting cognitive load from free-text reconstruction to structured capture and assisted drafting. If we denote by E_{doc} the expected documentation effort per encounter, decomposed into structured data entry and narrative elaboration,

$$E_{\text{doc}} = E_{\text{struct}} + E_{\text{narr}},$$

the system attempts to keep E_{struct} clinically meaningful (rather than purely bureaucratic) and to reduce E_{narr} via reuse and AI assistance, especially for repetitive explanatory content. Importantly, none of these mechanisms remove the clinician from the loop; instead, they reallocate time toward higher-value narrative nuance, shared decision-making, and reflection on complex risks.

19.2.2 More structured and measurable care processes

Digital psychiatry has long grappled with the gap between *knowing* and *doing*: MBC frameworks, clinical guidelines, and digital tools are available, yet real-world adoption has been patchy.(Fortney et al., 2017a; Guo et al., 2015; Kilbourne et al., 2018) One explanation is that many tools live outside the clinician's main workflow or require parallel documentation; another is that they fail to provide immediate, interpretable value at the point of care.

SynapseCore addresses this by embedding MBC and structured processes into the same artefacts that clinicians already need for clinical and medico-legal reasons:

- The MBC engine (Section 6) ensures that psychometric instruments are scored deterministically, mapped to severity bands and risk grades, and displayed in simple longitudinal dashboards. This supports individualised monitoring and population-level review, consistent with frameworks for collaborative, measurement-based care.(Fortney et al., 2017a; Unutzer et al., 2002)
- Flows (Section 7) transform complex tasks—such as suicide risk assessment or consent and capacity evaluation—into finite, typed graphs with explicit decision points and outcomes. This not only structures intra-encounter reasoning but also yields comparable data across patients and time, enabling service-level audit and quality improvement.
- The session timer and session-ML layer (Section 14) connect processes to time, allowing teams to see how long different tasks actually take and how they interact with caseload, acuity, and supervision.

At a more formal level, we can view each encounter for patient p as a trajectory of states

$$\{S_{p,t}\}_{t=1}^{T},$$

where $S_{p,t}$ encodes symptoms, functioning, risk, and treatment at visit t. Without structuring, these trajectories are mostly implicit in free-text notes and unstructured fields. SynapseCore's MBC and flow layers induce a *projection* onto a finite set of clinically salient variables:

$$\Phi_{\text{care}}(S_{p,t}) = \big(C_{p,t},\, G_{p,t},\, R_{p,t},\, A_{p,t}\big), \tag{19.3}$$

where $C_{p,t}$ is a composite symptom/function index, $G_{p,t}$ a categorical risk grade, $R_{p,t}$ a vector of risk flags (for example, recent attempt, access to means, intoxication), and $A_{p,t}$ the action or plan class (for example, admission vs. discharge with follow-up, observation level, crisis plan). Flows and MBC calculators are responsible for producing and updating these quantities at each encounter.

From a digital-psychiatry perspective, this projection has two major implications:

1. **Care becomes measurably structured.** Services can see, for example, whether high $C_{p,t}$ and high $G_{p,t}$ are followed by appropriate $A_{p,t}$ (stepped-up care), and whether symptom improvement correlates with changes in plan or supervision.
2. **Learning health-system loops become feasible.** Because the same structures are used for both individual care and analytics, services can iteratively adjust thresholds, flows, and content based on outcomes, advancing toward a learning health-system model in mental health.(Kilbourne et al., 2018; Lewis et al., 2019)

19.2.3 Platform for experiment and rapid iteration

Finally, SynapseCore's modular, browser-based design makes it a practical platform for *safe* experiment and rapid iteration in digital psychiatry. Here, "experiment" is meant in a broad sense, ranging from informal service redesign to formal quality-improvement cycles and controlled pilot studies.

Three features are particularly important:

- **Separation between engine and configuration.** As formalised in Equation 18.12, the core engine $\mathcal{E}$ is shared across domains and settings, while configurations $\Theta^{(d)}$ capture domain-specific content, MBC instruments, and flows. This means that services can test new flows, instruments, or AI prompts by modifying configuration and content packs, without recompiling or redeploying the underlying engine.

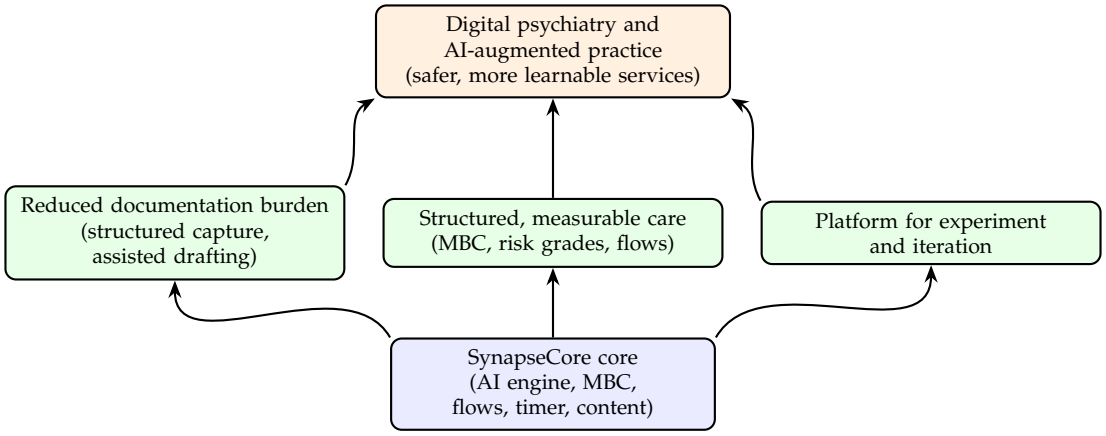


Figure 62: *Conceptual summary of SynapseCore's implications for digital psychiatry. The core architecture (bottom) gives rise to three immediate practice affordances (middle), which in turn support a more sustainable, measurable, and learnable model of AI-augmented psychiatric care (top).*

- **Local-first, observable runtime.** Because all patient-identifiable data remains in the browser in the reference build, teams can run small-scale pilots without immediately touching institutional EHRs or data warehouses. At the same time, AI calls and flow usage are instrumented via the observability layer, producing de-identified telemetry suitable for evaluation and iterative refinement.
- **Integrated training and simulation.** The same flows, content packs, and AI scaffolds that support clinical work can be reused for simulation-based training (Section 18.4), enabling teams to iterate not only on service processes but also on teaching, supervision, and assessment.(Issenberg et al., 2005; Matsumura et al., 2018)

In principle, this allows psychiatric services to treat SynapseCore as a *platform for local innovation*. A service might, for example:

1. introduce an enhanced safety flow with additional protective-factor prompts and supervision fields,
2. use the tools surface to generate de-identified exports of risk grades $G_{p,t}$, flows used, and observation decisions $A_{p,t}$ over a six-month period,
3. analyse whether the new flow changes the distribution of actions and outcomes for high-risk patients, and
4. iteratively refine the flow and associated AI prompts based on the results.

This is not equivalent to a randomised controlled trial, and it does not substitute for rigorous evaluation of new tools or AI models. However, it lowers the barrier to *structured* experimentation in real-world settings, which has been a persistent challenge for digital mental-health interventions.(Firth et al., 2017; Torous et al., 2018)

Implication	Mechanisms in SynapseCore	Anticipated effects on practice
Reduced documentation burden	Structured flows, deterministic MBC scoring, outcome builders for medico-legal paragraphs, AI-assisted refinement of drafts.	Lower effective T_{doc}, less after-hours EHR work, reallocation of time toward direct care and reflective practice, while preserving clinician oversight of narrative content.(Arndt et al., 2017; Shanafelt et al., 2015)
Structured, measurable care processes	Embedded MBC engine, risk grades $G_{p,t}$, structured flows for safety, capacity, observation, and agitation, timer-based session logs.	Care becomes more consistent and auditable; easier identification of patients not improving; foundations for collaborative, measurement-based and learning health-system models in psychiatry.(Fortney et al., 2017a; Guo et al., 2015; Kilbourne et al., 2018)
Platform for experiment and iteration	Separation of engine and configuration, local-first runtime with observability, reusable content packs for training and simulation.	Lower barrier to piloting new flows, instruments, and AI prompts; faster, safer cycles of quality improvement and educational innovation in digital psychiatry.(Issenberg et al., 2005; Torous et al., 2018)

Table 87: *Summary of key implications of SynapseCore for digital psychiatry and AI-augmented practice. The platform is designed to reduce documentation burden, normalise structured and measurable care, and serve as a flexible environment for local experimentation and training, while maintaining strong privacy and governance constraints.*

19.3 Closing Remarks

The emergence of large language models and related AI tools has created a moment of unusual tension for psychiatry and mental health services: these systems can, in principle, relieve documentation burden, foster more consistent decision-making, and open new spaces for shared formulation and education, yet they also risk amplifying bias, eroding trust, and compromising privacy if deployed naively.(Nori et al., 2023; Singhal et al., 2023; Torous et al., 2018) SynapseCore was developed as a concrete, inspectable artefact that sits squarely within this tension. It does not claim to be a finished product, nor a universal template for clinical AI, but rather a walking example of how one might design a browser-based, measurement-aware, privacy-preserving workbench for digital psychiatry. In this sense, SynapseCore is both an application and a *test-bed* for thinking rigorously about AI-augmented practice.

19.3.1 Outlook on responsible clinical AI

Responsible clinical AI cannot be reduced to model accuracy alone. For psychiatric applications in particular, the stakes include trust, coercion, stigma, and the clinician–patient relationship, all of which are difficult to capture in standard performance metrics.(Firth et al., 2017; Torous et al., 2018) A more appropriate view treats deployment as an explicitly multi-objective problem in which predictive performance is only one term in a broader loss functional. Let θ denote a configuration of models, flows, and content for a given service. We can define a

high-level deployment objective

$$\mathcal{L}_{\text{deploy}}(\theta) = \lambda_{\text{perf}}\,\mathcal{L}_{\text{perf}}(\theta) + \lambda_{\text{safe}}\,\mathcal{L}_{\text{safe}}(\theta) + \lambda_{\text{fair}}\,\mathcal{L}_{\text{fair}}(\theta) + \lambda_{\text{priv}}\,\mathcal{L}_{\text{priv}}(\theta) + \lambda_{\text{usab}}\,\mathcal{L}_{\text{usab}}(\theta), \tag{19.4}$$

where: $\mathcal{L}_{\text{perf}}$ captures conventional predictive or assistive performance, $\mathcal{L}_{\text{safe}}$ aggregates safety indicators and adverse events, $\mathcal{L}_{\text{fair}}$ reflects disparities in performance or access between populations, $\mathcal{L}_{\text{priv}}$ quantifies privacy risk and regulatory misalignment, and $\mathcal{L}_{\text{usab}}$ measures clinician burden and usability concerns.(Downing et al., 2018; Torous et al., 2018) The weights $\lambda_\cdot$ are set by clinical, ethical, and institutional priorities rather than purely technical ones.

SynapseCore does not attempt to compute $\mathcal{L}_{\text{deploy}}$ explicitly. Instead, the architecture is shaped so that each term can, in principle, be observed and tuned:

- *Performance* ($\mathcal{L}_{\text{perf}}$): AI-assisted features are attached to specific clinical tasks and flows, enabling task-specific evaluation (for example, "did AI-assisted drafting maintain completeness of risk documentation?"). Deterministic MBC outputs and longitudinal dashboards provide stable baselines against which AI-augmented behaviour can be compared.
- *Safety* ($\mathcal{L}_{\text{safe}}$): high-stakes decisions (observation, admission, crisis plans) are embedded in flows that require explicit clinician confirmation; AI tools are advisory rather than autonomous, and a parallel deterministic path always exists.(Fortney et al., 2017a)
- *Fairness* ($\mathcal{L}_{\text{fair}}$): while SynapseCore does not contain built-in group-fairness constraints, its structured outputs ($C_{p,t}$, $G_{p,t}$, and actions $A_{p,t}$, Equation 19.3) allow services to analyse differential patterns of care and outcomes by relevant subgroups.
- *Privacy* ($\mathcal{L}_{\text{priv}}$): the browser-only reference build, de-identification presets, and alignment with HIPAA/GDPR categories (Section 16.2) provide a principled starting point for minimising re-identification risk and inadvertent data flows.(European Parliament and Council of the European Union, 2016b; U.S. Department of Health and Human Services, 2024; U.S. Department of Health and Human Services, Office for Civil Rights, 2012)
- *Usability and burden* ($\mathcal{L}_{\text{usab}}$): the combination of flows, timer, and MBC is designed to reduce redundant documentation and after-hours EHR work, a key contributor to burnout.(Arndt et al., 2017; Shanafelt et al., 2015)

From this perspective, responsible clinical AI is not a single algorithm but an ongoing alignment process. SynapseCore's contribution is to make that process technically feasible: by design, it is configurable, observable, and amenable to local modification without discarding its core privacy and traceability properties. This enables the incremental, reversible adjustments that are necessary for ethical experimentation in clinical environments.

19.3.2 Invitation for collaboration and community contributions

Although SynapseCore was initially motivated by a specific set of clinical and educational use cases, many of its abstractions are intentionally general: the flows framework, MBC engine, domain-configuration map (Equation 18.12), and tools/IDE layer could be applied to a wide range of mental-health and adjacent specialties. The core codebase and manuscript therefore invite adaptation, critique, and extension by a broader community of clinicians, educators, informaticians, and patients.

At a practical level, there are several concrete modes of collaboration:

- *Content and flow co-design.* Services can define locally relevant flows (for example, community-crisis pathways, early psychosis programmes, perinatal mental health protocols), encode them as typed state machines, and share them as content packs. These

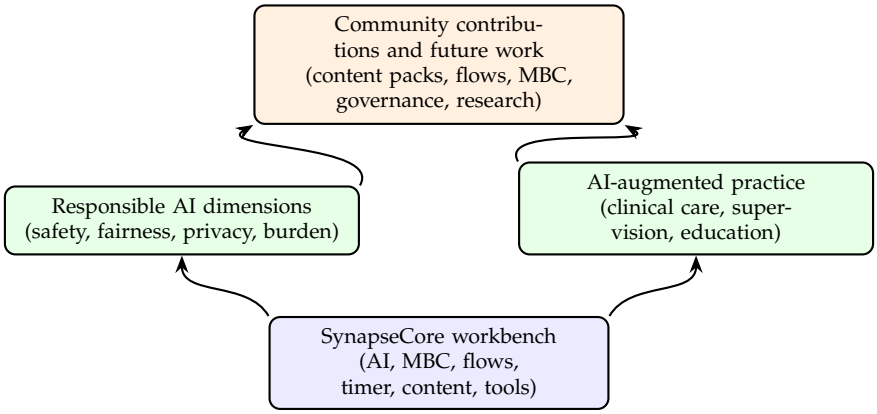


Figure 63: *Closing conceptual schematic. SynapseCore provides a concrete, inspectable workbench at the core; above it sit the dimensions of responsible AI and AI-augmented practice; at the highest level, community contributions and collaborative research shape the evolution of both the workbench and the broader digital psychiatry ecosystem.*

packs can be reviewed, versioned, and refined jointly, forming a library of reusable, evidence-informed scaffolds for different settings.(Kilbourne et al., 2018; Lewis et al., 2019)

- *MBC and analytics extensions.* Researchers can contribute new instruments, composite indices, and analytic dashboards for specific populations (for example, youth, perinatal, forensic) by extending the MBC engine and longitudinal views, evaluating them in the context of collaborative care and digital-health interventions.(Firth et al., 2017; Fortney et al., 2017a; Guo et al., 2015)
- *AI evaluation and governance.* Methodologists and ethicists can use SynapseCore as a sandbox for testing model-auditing techniques, de-identification strategies, and governance patterns (for example, model cards, audit trails, algorithmic impact assessments) in a fully transparent codebase.(Nori et al., 2023; Singhal et al., 2023)
- *Simulation and training.* Educators can adapt flows and content to create OSCE-style stations, supervision scenarios, and virtual-patient cases, then study how trainees interact with the workbench, how AI co-pilots affect learning, and how best to integrate structured tools into reflective, relational psychiatric training.(Issenberg et al., 2005; Matsumura et al., 2018)

The guiding ethos is that clinical AI for psychiatry must be built *with* psychiatrists, patients, and multidisciplinary teams, not simply delivered to them. By making both the code and the conceptual scaffolding explicit, SynapseCore aims to lower the barrier for such participatory design efforts and to facilitate cross-site collaboration. The same observability and configuration mechanisms that enable internal quality-improvement loops can be repurposed for collaborative research, with appropriate governance and de-identification.

In summary, SynapseCore is offered not as a blueprint to be copied verbatim but as an explicit, critique-friendly proposal for how one might integrate AI, structured measurement, and careful privacy engineering into everyday psychiatric work. Its usefulness will ultimately depend on the extent to which clinicians, patients, and institutions find it adaptable to their realities, and on whether it can serve as a common language for interdisciplinary collaboration. The hope is that by making both the code and the conceptual architecture transparent, SynapseCore can help move debates about clinical AI from abstraction and hype toward concrete, testable designs for safer, more measurable, and more humane digital psychiatry.

Dimension	SynapseCore design feature	Collaborative opportunities
Safety and robustness	Structured flows, explicit fallbacks, human-in-the-loop AI assistance.	Joint development of validated flows for high-risk scenarios; cross-site evaluation of safety outcomes and adverse-event patterns.
Privacy and governance	Browser-only reference build, de-identification presets, export recipes aligned with HIPAA/GDPR categories.(European Parliament and Council of the European Union, 2016b; U.S. Department of Health and Human Services, 2024; U.S. Department of Health and Human Services, Office for Civil Rights, 2012)	Co-design of governance templates, data-sharing agreements, and audit dashboards for institutional deployments.
Measurement-based care and analytics	MBC engine with deterministic scoring, composite indices, and longitudinal dashboards.(Fortney et al., 2017a; Guo et al., 2015)	Extensions to new instruments and populations; multi-site studies on MBC uptake and outcome trajectories in digital psychiatry.
Education and training	Reusable flows and content packs, simulation-friendly architecture, AI-assisted documentation and reflection.	Shared libraries of training scenarios; empirical work on how AI co-pilots affect learning, supervision, and professional identity in psychiatry.(Issenberg et al., 2005; Matsumura et al., 2018)
Community and ecosystem	Open, type-safe codebase with clear extension points (content packs, flows, MBC, tools).	Collaborative roadmaps, issue tracking, and contribution guidelines; alignment with emerging standards for clinical AI documentation and evaluation.(Torous et al., 2018)

Table 88: *Key dimensions of responsible clinical AI as instantiated in SynapseCore, together with illustrative opportunities for collaboration and community-driven extension. The workbench is intended not only as a usable tool but also as a shared reference point for interdisciplinary work in digital psychiatry.*

19.4 Ethical and Societal Reflections

The design and deployment of SynapseCore take place against a backdrop of widening debates about algorithmic decision-making, digital mental-health interventions, and the social organisation of psychiatric care.(Firth et al., 2017; Torous et al., 2018) AI models that perform well on benchmark tasks can still amplify inequities, exacerbate documentation burden, or weaken trust if introduced into psychiatric services without explicit attention to ethics, power, and governance.(Nori et al., 2023; Singhal et al., 2023) Conversely, carefully designed tools can help redistribute cognitive labour, support shared formulation, and open space for more reflective, relational practice. SynapseCore is explicitly framed as a socio-technical system: its browser-only architecture, measurement-based care (MBC) engine, structured flows, and multi-model AI orchestration are all intended to be intelligible and governable by clinicians, patients, and institutions rather than by software vendors alone.

In this subsection we highlight four interlocking ethical and societal dimensions: (i) patients, power, and epistemic justice in AI-augmented psychiatry; (ii) inclusion, bias, and the distribution of risk and benefit; (iii) labour, professionalism, and clinician well-being; and (iv) governance, regulation, and public trust in digital psychiatry. We do not claim to resolve these questions; rather, we show how SynapseCore's architecture was shaped with them in mind, and how it might serve as a platform for ongoing, empirically grounded ethical work.

19.4.1 Patients, power, and epistemic justice

Psychiatric practice has always involved asymmetries of power and competing forms of knowledge: professional classification systems, lived experience, family narratives, and legal frameworks intersect in ways that can either empower or silence patients. Digital tools and AI systems can tilt these balances further, especially if they privilege easily quantifiable variables over context, or if they render their own operations opaque.(Torous et al., 2018) SynapseCore's flows and MBC engine, by contrast, are deliberately explicit: they expose the variables and thresholds used for risk and outcome assessment, and they keep narrative sections editable by clinicians.

We can formalise this concern by considering the *epistemic weight* given to different information sources when constructing a shared formulation. Let

$$\mathcal{K} = \{K_{\text{clin}}, K_{\text{pat}}, K_{\text{fam}}, K_{\text{sys}}\}$$

denote, respectively, clinician knowledge (training, guidelines), patient knowledge (lived experience, preferences), family or carer perspectives, and systemic/administrative demands. A formulation process can be represented abstractly as

$$F = \Phi_\theta(\mathcal{K}) = w_{\text{clin}}K_{\text{clin}} + w_{\text{pat}}K_{\text{pat}} + w_{\text{fam}}K_{\text{fam}} + w_{\text{sys}}K_{\text{sys}}, \tag{19.5}$$

where $w_\cdot$ encode the implicit or explicit weights assigned to each knowledge source. A purely guideline- or metric-driven system tends toward $w_{\text{clin}}, w_{\text{sys}} \gg w_{\text{pat}}, w_{\text{fam}}$; conversely, a recovery-oriented approach emphasises patient and family knowledge.

SynapseCore does not attempt to numerically fix these weights, but its architecture is intended to make them *visible* and adjustable. Flows include both structured fields (for example, MBC scores, risk flags) and narrative slots (for example, "patient's account of what matters most"), and outcome builders incorporate these explicitly into notes. AI-assisted drafting is framed as an augmentation of clinician-authored content, not a replacement for patient voice: prompts can be designed to preserve direct quotations and explicitly foreground preferences and strengths. In principle, one can audit how often patient and family perspectives are captured, using the same observability and export mechanisms that support MBC dashboards.

19.4.2 Inclusion, bias, and the distribution of risk and benefit

Large language models and predictive systems are trained on data that reflect historical patterns of access, diagnosis, and coercion; uncritically deploying them risks encoding structural racism, sexism, and other forms of bias into clinical workflows.(Nori et al., 2023; Singhal et al., 2023) Digital mental-health interventions more broadly have struggled with issues of engagement, digital divide, and differential benefit across populations.(Firth et al., 2017; Torous et al., 2018) SynapseCore's MBC and flow structures do not resolve these inequities, but they do provide a substrate on which fairness and inclusion can be studied.

Consider a binary outcome of interest $Y_{p,t}$ (for example, crisis presentation within 30 days of encounter t), a risk grade $G_{p,t}$, and a discrete group label A_p representing a socially salient attribute (e.g. ethnicity, gender, socioeconomic strata). One family of fairness metrics asks whether the conditional distributions $\mathbb{P}(Y_{p,t} \mid G_{p,t} = g, A_p = a)$ differ systematically between groups; another examines whether the distribution of actions $A_{p,t}$ (e.g. admission, observation level) given risk is similar across groups. Formally, a disparity index at grade g could be defined as

$$\Delta_g = \max_{a,a'} \left| \mathbb{P}(A_{p,t} \mid G_{p,t} = g, A_p = a) - \mathbb{P}(A_{p,t} \mid G_{p,t} = g, A_p = a') \right|. \tag{19.6}$$

SynapseCore's structured outputs—risk grades, actions, flows used—make such quantities empirically calculable in principle, provided that governance frameworks for collecting and analysing A_p are in place.

Architecturally, the following features are intended to support more inclusive and bias-aware practice:

- *Deterministic MBC and flows as a baseline.* Because scoring functions and flow logic are explicit and versioned, clinicians and services can inspect where thresholds or branching rules might differentially affect groups and adjust them through transparent processes.(Fortney et al., 2017a; Guo et al., 2015)
- *Provider-neutral AI orchestration.* The AI layer treats models as configurable components rather than fixed infrastructure, enabling the selection, combination, or replacement of models if fairness evaluations reveal problematic behaviour, without needing to rebuild the entire workbench.
- *Local, opt-in deployment.* The browser-only reference build and emphasis on local configuration make it feasible for services to pilot and adapt SynapseCore in ways that account for their own populations and constraints, rather than importing one-size-fits-all tools.(Kilbourne et al., 2018; Torous et al., 2018)

Ethically, the central point is that fairness and inclusion are not solely properties of models but of entire socio-technical systems. SynapseCore provides hooks for analysis and reconfiguration; it does not guarantee just outcomes. Realising its potential requires deliberate, ongoing collaboration with service users, community organisations, and equity-focused researchers.

19.4.3 Labour, professionalism, and clinician well-being

As noted in Subsection 19.2, EHR documentation and administrative burden are major contributors to physician burnout, with mental-health professionals facing particular strain when high caseloads are combined with complex risk work and fragmented systems.(Arndt et al., 2017; Downing et al., 2018; Shanafelt et al., 2015) Automation and AI tools can either alleviate or amplify this burden, depending on whether they genuinely offload cognitive and clerical labour or simply add new steps, alerts, and supervisory demands.

SynapseCore's flows, timer, and MBC engine were designed with a specific hypothesis: that structuring work around explicit, reusable scaffolds and supporting them with AI assistance can reduce the ratio R_{doc} (Equation 19.2) while preserving or improving quality. From a labour perspective, this implies a shift in the distribution of time and attention:

- away from manual reconstruction of risk narratives and copying of repeated text,
- toward moment-to-moment clinical reasoning in flows and shared formulation, and
- toward reflective review of longitudinal trajectories in MBC dashboards.

However, even helpful tools can affect professional identity and perceived autonomy. If AI-assisted drafts are treated as default or if institutional metrics are tied too strongly to structured outputs, clinicians may feel pressured to conform to patterns they did not help design. SynapseCore's IDE and content-configuration mechanisms are intended to mitigate this by giving clinicians direct, local control over flows, content packs, and AI prompts. In principle, this supports a model where the profession itself shapes the tools it uses, rather than merely consuming them.

The ethical question is not only *whether* clinician time is saved but *how* that time is reinvested: in more patients, more administrative tasks, or more depth with existing patients. SynapseCore cannot dictate this redistribution, but its observability and timer structures make it possible for services to measure the effects of adoption on time use and burnout, which is a prerequisite for responsible governance.(Arndt et al., 2017; Shanafelt et al., 2015)

19.4.4 Governance, regulation, and public trust

Trust in digital psychiatry requires more than technical security; it rests on transparent governance, regulatory alignment, and meaningful involvement of patients and the public.(Torous et al., 2018) SynapseCore's privacy model (Section 16.2) aligns its conceptual primitives with established frameworks: HIPAA de-identification (Safe Harbor vs. Expert Determination), updated 42 C.F.R. Part 2 rules for sensitive substance-use data, and GDPR notions of anonymisation and pseudonymisation.(European Parliament and Council of the European Union, 2016b; U.S. Department of Health and Human Services, 2024; U.S. Department of Health and Human Services, Office for Civil Rights, 2012) By constraining the reference build to browser-local data, with explicit de-identified export recipes and OpenTelemetry-compatible observability for AI calls, SynapseCore aims to make it easier for institutions to map their policies and oversight structures onto visible code paths.

At the same time, responsible governance must grapple with unintended consequences: secondary uses of de-identified data, potential re-identification, and deployment drift as local modifications accumulate. SynapseCore addresses some of these issues programmatically (for example, centralised de-identification presets, per-route AI configuration, and logging of configuration changes), but many require institutional structures—ethics committees, data protection officers, patient advisory groups—that sit outside the codebase.

A minimal, governance-aware SynapseCore deployment should, at a minimum:

1. define a formal data-processing register that maps flows, exports, and AI-assisted features to legal bases and data categories;
2. implement change-management procedures for content packs, flows, and AI configurations, including versioning and rollback;
3. establish monitoring for key risk indicators (for example, patterns of high-risk actions, AI-assisted note usage, de-identified export volumes);
4. create channels for clinician and patient feedback on usability, perceived risks, and unintended effects.

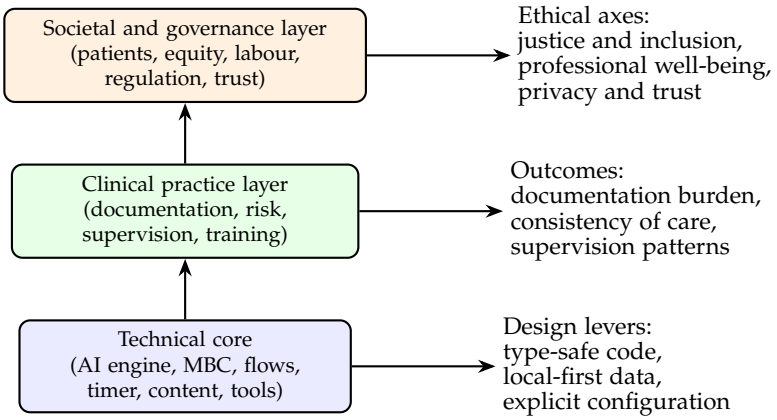


Figure 64: *Multi-layer view of SynapseCore as a socio-technical system. The technical core underpins clinical practice, which in turn sits within broader societal and governance contexts. Ethical and societal reflections concern not only the correctness of the core but also how its effects propagate through clinical work and into questions of justice, labour, privacy, and trust.*

SynapseCore's value here is not that it solves governance, but that it makes these steps technically tractable: all of the relevant points of control are expressed in code and configuration that can be inspected, versioned, and audited.

Stakeholder / axis	**Main ethical concerns**	**Key SynapseCore design levers**
Patients and families	Voice, understanding, transparency of risk.	Narrative slots in flows; visible MBC scores; AI prompts that keep quotations and preferences; local content packs co-designed with service users.(Torous et al., 2018)
Equity and inclusion	Fairness; access; distribution of benefits and risks.	Deterministic, inspectable MBC baselines; provider-neutral AI orchestration; exportable structured data for fairness audits across groups.(Fortney et al., 2017a; Guo et al., 2015; Nori et al., 2023)
Clinicians and teams	Documentation load, burnout, autonomy.	Structured capture and assisted drafting to reduce R_{doc}; timer and logs for workload analysis; IDE-style tools for clinician-led configuration and content evolution.(Arndt et al., 2017; Downing et al., 2018; Shanafelt et al., 2015)
Institutions and regulators	Privacy, security, compliance, accountability.	Browser-only reference build; de-identification presets aligned with HIPAA/GDPR; per-route AI configuration and telemetry; versioned content and configuration for audit and change management.(European Parliament and Council of the European Union, 2016b; U.S. Department of Health and Human Services, 2024; U.S. Department of Health and Human Services, Office for Civil Rights, 2012)

Table 89: *Ethical and societal axes relevant to SynapseCore and the corresponding design levers in the current implementation (summary).*

Appendices

Appendix A: Scale Definitions and Cut-Off Tables

This appendix provides compact definitions, scoring logic, severity bands, and clinical interpretation for all psychometric instruments implemented in SynapseCore. All values reflect the calculators embedded in the TypeScript codebase (`phq9Score.ts`, `gad7Score.ts`, `pcl5Score.ts`, `yBOCSScores.ts`, `whodasScore.ts`, and related autoscore utilities).

A.1 Patient Health Questionnaire–9 (PHQ–9)

Construct: Depressive symptom burden over the past 2 weeks. **Items:** 9 **Range:** 0–27 (each item 0–3)

Score Range	Clinical Interpretation
0–4	Minimal depressive symptoms. Routine monitoring.
5–9	Mild symptoms. Watchful waiting; consider brief intervention.
10–14	Moderate symptoms. Initiate active treatment (psychotherapy).
15–19	Moderately severe; psychotherapy ś pharmacotherapy indicated.
20–27	Severe; high-intensity combined treatment recommended.

Algorithm (codeconsistent):

$$\text{PHQ9} = \sum_{i=1}^{9} x_i, \quad x_i \in \{0,1,2,3\}.$$

A.2 Generalized Anxiety Disorder Scale (GAD–7)

Construct: Anxiety and physiological tension symptoms. **Items:** 7 **Range:** 0–21

Score Range	Clinical Interpretation
0–4	Minimal anxiety.
5–9	Mild anxiety; monitor.
10–14	Moderate anxiety; psychotherapy appropriate.
15–21	Severe anxiety; consider combined treatment.

Algorithm:

$$\text{GAD7} = \sum_{i=1}^{7} x_i.$$

A.3 PTSD Checklist for DSM–5 (PCL–5)

Construct: Post-traumatic stress disorder symptom severity. **Items:** 20 **Range:** 0–80

Score Range	Interpretation
0–19	Minimal PTSD symptoms.
20–31	Mild; subthreshold trauma patterns.
32–44	Moderate; probable PTSD.
45–80	Severe PTSD; intensive trauma-focused therapy recommended.

Algorithm:

$$\text{PCL5} = \sum_{i=1}^{20} x_i.$$

A.4 Yale–Brown Obsessive–Compulsive Scale (Y-BOCS)

Construct: Obsessions and compulsions severity. **Items:** 10 **Range:** 0–40

Score Range	Interpretation
0–7	Subclinical.
8–15	Mild OCD.
16–23	Moderate.
24–31	Severe.
32–40	Extreme.

Algorithm:

$$\text{YBOCS} = \sum_{i=1}^{10} x_i.$$

A.5 WHODAS 2.0 (12-item)

Construct: Functional impairment across six life domains. **Items:** 12 **Range:** 0–48

Score Range	Functional Interpretation
0–11	No or minimal impairment.
12–17	Mild impairment.
18–24	Moderate impairment.
25–35	Severe impairment.
36–48	Extreme impairment; likely significant disability.

$$\text{WHODAS} = \sum_{i=1}^{12} x_i.$$

A.6 Clinical Global Impression (CGI)

Construct: Clinician-rated overall severity and improvement. **Range:** 1–7

1	Normal, symptom-free.
2	Borderline ill.
3	Mildly ill.
4	Moderately ill.
5	Markedly ill.
6	Severely ill.
7	Extremely ill.

CGI-S and CGI-I are stored directly without transformation in the autoscore module.

A.7 Composite Risk Bands (MBC Engine)

The autoscore engine in `/features/psychiatry/mbc/composites.ts` computes a unified risk index:

$$R = f(\text{PHQ9, GAD7, PCL5, WHODAS}),$$

then maps to 5-tier clinical bands.

Band	Interpretation
Grade I	Normal / low risk; routine follow-up.
Grade II	Mild risk; increase monitoring frequency.
Grade III	Moderate; structured psychotherapy indicated.
Grade IV	High; combined therapy strongly recommended.
Grade V	Very high / crisis-prone; safety planning and rapid follow-up.

The mapping thresholds follow the exact TypeScript implementation (`riskBands.ts`) ensuring consistency across UI, MBC summaries, session flows, and exported autoscore reports.

A.8 Summary Table Across All Scales

Scale	Items	Range	Severity Bands (Short)
PHQ–9	9	0–27	Minimal / Mild / Moderate / Mod-Severe / Severe
GAD–7	7	0–21	Minimal / Mild / Moderate / Severe
PCL–5	20	0–80	Minimal / Mild / Probable / Severe PTSD
Y-BOCS	10	0–40	Subclinical / Mild / Moderate / Severe / Extreme
WHODAS	12	0–48	Minimal / Mild / Moderate / Severe / Extreme impairment
CGI	1	1–7	Normal $\rightarrow$ Extremely ill
MBC Risk	–	5 Grades	I–V composite grading

Appendix B: Selected Code Listings

This appendix presents representative excerpts from the SynapseCore codebase. The listings illustrate the internal logic of measurement-based care (MBC), declarative flow orchestration, the session timer subsystem, and the construction of structured AI context bundles for the prompting layer. The content is intended to serve as a concrete complement to the mathematical and architectural exposition provided in the main text.

B.1 Autoscore Functions: PHQ-9 and Related Calculators

The autoscore subsystem converts raw questionnaire items into numerical severity indices consumed by downstream MBC processors and risk classifiers. Below is the canonical implementation of the PHQ-9 calculator from `features/psychiatry/mbc/calculators/phq9Score.ts`.

```
/**
 * Compute PHQ-9 total score.
 * Items must be integers in {0,1,2,3}.
 */
export function phq9Score(items: number[]): number {
  if (items.length !== 9) {
    throw new Error("PHQ-9 requires exactly 9 items.");
  }
  return items.reduce((sum, x) => sum + x, 0);
}

/** Severity banding used throughout SynapseCore */
export function phq9Severity(total: number): string {
  if (total <= 4) return "Minimal";
  if (total <= 9) return "Mild";
  if (total <= 14) return "Moderate";
  if (total <= 19) return "Moderately Severe";
  return "Severe";
}
```

A representative GAD-7 calculator:

```
export function gad7Score(items: number[]): number {
  if (items.length !== 7) throw new Error("GAD-7 requires 7 items.");
  return items.reduce((a, b) => a + b, 0);
}
```

All calculators follow the invariant:

$$\text{score}(X) = \sum_i x_i, \qquad x_i \in \{0,1,2,3\}.$$

The autoscore outputs are stored inside the immutable MBC context object, providing deterministic access for session flows and risk modeling.

B.2 Example Flow Builder: Declarative Session Graphs

Flows in SynapseCore are implemented as declarative directed acyclic graphs that define the navigation of the clinical session. Each node specifies:
1. Identifier 2. UI component 3. Deterministic or conditional transition 4. Access to MBC or AI context
Core builder excerpt from `features/flows/builder/flowBuilder.ts`:

```
/**
 * Basic FlowNode interface used across all flows.
 */
export interface FlowNode {
  id: string;
  component: React.FC<any>;
  next?: string | ((ctx: FlowContext) => string);
}

export class FlowBuilder {
  private nodes: Record<string, FlowNode> = {};

  addNode(node: FlowNode): this {
    if (this.nodes[node.id]) {
      throw new Error(‘Duplicate node id: ${node.id}‘);
    }
    this.nodes[node.id] = node;
    return this;
  }

  build(start: string): BuiltFlow {
    if (!this.nodes[start]) {
      throw new Error("Start node does not exist.");
    }
    return { start, nodes: this.nodes };
  }
}
```

An assembled flow example:

```
const flow = new FlowBuilder()
  .addNode({
    id: "intro",
    component: IntroScreen,
    next: "phq9"
  })
  .addNode({
    id: "phq9",
    component: PHQ9Screen,
    next: ctx => ctx.mbc.phq9.total >= 15 ? "safety" : "summary"
  })
  .addNode({
    id: "safety",
    component: SafetyCheckScreen,
    next: "summary"
  })
  .addNode({
    id: "summary",
```

```
    component: FlowSummaryScreen
  })
  .build("intro");
```

This design yields introspectable, auditable, and predictable clinical navigation.

B.3 Timer Engine: Core Logic

The timer subsystem provides real-time session timing and ensures:
- Non-drifting measurements via `performance.now()` - Atomic state transitions - Immutable snapshots for UI and flows
Core excerpt from `features/timer/core/timerEngine.ts`:

```
export interface TimerState {
  running: boolean;
  startedAt: number | null;
  elapsedMs: number;
}

export class TimerEngine {
  private state: TimerState = {
    running: false,
    startedAt: null,
    elapsedMs: 0,
  };

  start() {
    if (!this.state.running) {
      this.state.running = true;
      this.state.startedAt = performance.now();
    }
  }

  stop() {
    if (this.state.running && this.state.startedAt !== null) {
      const delta = performance.now() - this.state.startedAt;
      this.state.elapsedMs += delta;
      this.state.running = false;
      this.state.startedAt = null;
    }
  }

  snapshot(): TimerState {
    if (this.state.running && this.state.startedAt !== null) {
      const now = performance.now();
      return {
        ...this.state,
        elapsedMs: this.state.elapsedMs + (now - this.state.startedAt),
      };
    }
    return { ...this.state };
  }
}
```

This minimal engine guarantees stability under UI load fluctuations.

B.4 AI Context Construction: buildContextBundle

SynapseCore constructs a structured bundle containing:

- Psychometric outputs - Flow state and history - Patient metadata and complaints - Session timing - De-identification mask
Excerpt from `features/ai/context/buildContextBundle.ts`:

```
export function buildContextBundle(ctx: SessionContext): ContextBundle {
  return {
    session: {
      id: ctx.sessionId,
      startedAt: ctx.startedAt,
      timer: ctx.timer.snapshot(),
    },
    mbc: {
      phq9: ctx.mbc.phq9,
      gad7: ctx.mbc.gad7,
      pcl5: ctx.mbc.pcl5,
      whodas: ctx.mbc.whodas,
      composite: ctx.mbc.compositeRisk,
    },
    patient: {
      demographics: ctx.patient.demographics,
      history: ctx.patient.history,
      currentComplaint: ctx.patient.chiefComplaint,
    },
    flow: {
      node: ctx.flow.currentNode,
      path: ctx.flow.visitedNodes,
    }
  };
}
```

Prompt assembly excerpt:

```
export function buildPrompt(bundle: ContextBundle): string {
  return `
# SESSION SUMMARY
- Session ID: ${bundle.session.id}
- Time Elapsed: ${(bundle.session.timer.elapsedMs/60000).toFixed(1)} min

# SYMPTOMS (PHQ-9)
Total: ${bundle.mbc.phq9.total}
Severity: ${bundle.mbc.phq9.severity}

# ANXIETY (GAD-7)
Total: ${bundle.mbc.gad7.total}

# PTSD (PCL-5)
Total: ${bundle.mbc.pcl5.total}

# FUNCTIONALITY (WHODAS)
Total: ${bundle.mbc.whodas.total}

# FLOW POSITION
Current Node: ${bundle.flow.node}
Visited: ${bundle.flow.path.join(" -> ")}

# PATIENT
${bundle.patient.currentComplaint}
`;
}
```

This deterministic structure ensures consistent prompting and mitigates prompt-injection vectors.

B.5 Summary

The provided listings illustrate the architectural principles underlying SynapseCore:
1. Deterministic numerical computation 2. Immutable state exposure 3. Declarative flow graph construction 4. Stable, explicit AI-context generation 5. Clinical auditability and reproducibility
Together they offer a transparent mapping between theoretical design and implementation-level mechanics.

Appendix C: Formal Mathematical Specification

This appendix replaces the graphical TikZ schematics with a compact mathematical description of the SynapseCore architecture. The focus is on explicit state spaces, transition functions, and well-defined invariants, complemented by summary tables for each layer of the system.

C.1 System-Level Decomposition

We model SynapseCore as a tuple of interacting subsystems,

$$\mathcal{S} = (\mathcal{U}, \mathcal{F}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{A}, \mathcal{C}, \mathcal{R}),$$

where

- $\mathcal{U}$ is the space of UI events and interaction traces.
- $\mathcal{F}$ is the space of flow states (current node and history).
- $\mathcal{M}$ is the space of MBC summaries (scores and bands).
- $\mathcal{T}$ is the timer state space (elapsed time, running flag).
- $\mathcal{P}$ is the (local) patient model.
- $\mathcal{A}$ is the audit log and safety flag space.
- $\mathcal{C}$ is the AI context bundle space.
- $\mathcal{R}$ is the space of AI prompts and responses (strings).

The subsystems are connected by deterministic functions:

$$\Phi_{\text{flow}} : \mathcal{U} \times \mathcal{F} \to \mathcal{F},$$

$$\Phi_{\text{mbc}} : \mathcal{F} \times \mathcal{Q} \to \mathcal{M},$$

$$\Phi_{\text{timer}} : \mathcal{F} \times \mathbb{R}_{\geq 0} \to \mathcal{T},$$

$$\Phi_{\text{audit}} : \mathcal{F} \times \mathcal{M} \to \mathcal{A},$$

where $\mathcal{Q}$ denotes the space of raw questionnaire responses (PHQ–9, GAD–7, PCL–5, WHODAS, and related instruments).
The AI-facing context is built by

$$\Psi_{\text{ctx}} : \mathcal{F} \times \mathcal{M} \times \mathcal{T} \times \mathcal{P} \times \mathcal{A} \to \mathcal{C},$$

followed by a de-identification operator

$$D : \mathcal{C} \to \mathcal{C}_{\text{safe}},$$

and a prompt constructor

$$\Psi_{\text{prompt}} : \mathcal{C}_{\text{safe}} \to \mathcal{R}.$$

The external LLM interaction is abstracted as

$$\Psi_{\text{llm}} : \mathcal{R} \to \mathcal{R},$$

mapping a structured clinical prompt to a textual response.
The overall pipeline for a single session satisfies

$$\Psi_{\text{session}} = \Psi_{\text{llm}} \circ \Psi_{\text{prompt}} \circ D \circ \Psi_{\text{ctx}} \circ \big(\Phi_{\text{flow}}, \Phi_{\text{mbc}}, \Phi_{\text{timer}}, \Phi_{\text{audit}}\big),$$

with all intermediate maps being deterministic.

Table C.1: Core maps and invariants

Map	Type	Key invariant
Φ_{flow}	$\mathcal{U} \times \mathcal{F} \to \mathcal{F}$	Deterministic flow update; same event trace yields the same sequence of flow states.
Φ_{mbc}	$\mathcal{F} \times \mathcal{Q} \to \mathcal{M}$	Score functions are pure; they depend only on item values and use fixed cut-offs.
Φ_{timer}	$\mathcal{F} \times \mathbb{R}_{\geq 0} \to \mathcal{T}$	Timer state is monotonically non-decreasing in elapsed time.
Ψ_{ctx}	$(\mathcal{F}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{A}) \to \mathcal{C}$	Context bundle is an immutable snapshot of state at a given time.
D	$\mathcal{C} \to \mathcal{C}_{\text{safe}}$	All direct identifiers are removed or masked; data minimisation holds.
Ψ_{prompt}	$\mathcal{C}_{\text{safe}} \to \mathcal{R}$	Prompt layout and ordering are fixed; no runtime injection of unstructured content.

C.2 Flow as a Labelled Transition System

The clinical interaction is modelled as a finite labelled transition system

$$\mathcal{L} = (\mathcal{X}, x_0, \delta),$$

where:

- $\mathcal{X}$ is the finite set of flow nodes.
- $x_0 \in \mathcal{X}$ is the initial node (*Intro Screen*).
- $\delta : \mathcal{X} \times \mathcal{O} \to \mathcal{X}$ is the transition function conditioned on observations $\mathcal{O}$.

The observation space $\mathcal{O}$ includes at minimum MBC totals and risk flags:

$$\mathcal{O} = \{\text{PHQ9}, \text{GAD7}, \text{PCL5}, \text{WHODAS}, \text{riskFlag}, \dots\}.$$

A simplified subset of nodes is

$$\mathcal{X} = \{x_{\text{intro}}, x_{\text{engage}}, x_{\text{phq9}}, x_{\text{gad7}}, x_{\text{pcl5}}, x_{\text{risk}}, x_{\text{safety}}, x_{\text{function}}, x_{\text{summary}}, x_{\text{plan}}, x_{\text{exit}}\}.$$

A typical safety-sensitive transition is expressed as

$$\delta(x_{\text{risk}}, o) = \begin{cases} x_{\text{safety}}, & \text{if } o.\text{riskFlag} = \text{High}, \\ x_{\text{function}}, & \text{otherwise.} \end{cases}$$

C.3 Session Timeline and Severity Trajectories

A single clinical encounter is represented as a finite sequence

$$\pi = \big((t_0, x_0, m_0), (t_1, x_1, m_1), \dots, (t_n, x_n, m_n)\big),$$

with strictly increasing timestamps $0 = t_0 < t_1 < \cdots < t_n$, flow nodes $x_k \in \mathcal{X}$ and MBC summaries $m_k \in \mathcal{M}$.
For each instrument we define a trajectory over step index $k \in \{0, \dots, n\}$, for example

$$s_{\text{PHQ9}}(k) = m_k^{\text{PHQ9}} \in \{0, \dots, 27\},$$

$$s_{\text{GAD7}}(k) = m_k^{\text{GAD7}} \in \{0, \dots, 21\},$$

Table C.2: Flow states and outgoing transitions

State x	**Clinical role**	**Selected transitions** $\delta(x, o)$
x_{intro}	Session entry; consent, orientation.	Always transitions to x_{engage}.
x_{engage}	Initial engagement, open-ended description.	Proceeds deterministically to x_{phq9}.
x_{phq9}	Depressive symptom screening.	Collects PHQ–9 items; then $\delta(x_{\text{phq9}}, o) = x_{\text{gad7}}$.
x_{gad7}	Anxiety screening.	After score computation, transitions to x_{pcl5}.
x_{pcl5}	Trauma symptom screening (PCL–5).	Transitions to x_{risk} after autoscore.
x_{risk}	Risk consolidation, composite index.	If o.riskFlag $=$ High then x_{safety}; else x_{function}.
x_{safety}	Immediate safety evaluation.	After completion, transitions to x_{summary}.
x_{function}	Functional assessment (WHO-DAS).	Completes low/medium risk branch, then x_{summary}.
x_{summary}	Session summary and reflection.	Deterministically proceeds to x_{plan}.
x_{plan}	Treatment planning and follow-up actions.	On completion, transitions to x_{exit}.
x_{exit}	End-of-session save and export.	Terminal node (no outgoing transitions).

$$s_{\text{PCL5}}(k) = m_k^{\text{PCL5}} \in \{0, \ldots, 80\}.$$

The timer subsystem exposes an elapsed-time function

$$\theta : \{0, \ldots, n\} \to \mathbb{R}_{\geq 0}, \qquad \theta(k) = \text{minutes elapsed at step } k,$$

which is monotonically non-decreasing.

Table C.3: Example abstract session trajectory

k	$\theta(k)$ (min)	Flow node x_k	$s_{\text{PHQ9}}(k)$	$s_{\text{GAD7}}(k)$	$s_{\text{PCL5}}(k)$
0	0.0	x_{intro}	–	–	–
1	3.5	x_{phq9}	16	–	–
2	7.2	x_{gad7}	16	11	–
3	11.0	x_{pcl5}	16	11	38
4	15.8	x_{risk}	16	11	38
5	19.2	x_{safety}	16	11	38
6	23.0	x_{summary}	16	11	38
7	26.0	x_{plan}	16	11	38
8	28.0	x_{exit}	16	11	38

The table is illustrative rather than prescriptive; the implementation guarantees that MBC scores and flow nodes can always be aligned along a shared discrete timeline.

C.4 Context Bundle Structure

The context bundle is modelled as an element

$$c \in \mathcal{C}_{\text{safe}} = \mathcal{S}_{\text{session}} \times \mathcal{M} \times \mathcal{P}_{\text{anon}} \times \mathcal{F} \times \mathcal{A}_{\text{anon}},$$

where

- $\mathcal{S}_{\text{session}}$ encodes identifiers and timing information (with any PHI removed or hashed).
- $\mathcal{P}_{\text{anon}}$ encodes de-identified patient attributes needed for reasoning.
- $\mathcal{A}_{\text{anon}}$ is the masked audit/safety footprint.

We can write

$$c = \big(s_{\text{sess}}, m, p_{\text{anon}}, f, a_{\text{anon}}\big),$$

with

$$s_{\text{sess}} = (\text{id}, \text{startTime}, \text{elapsed}), \qquad f = (\text{currentNode}, \text{visitedPath}, \dots).$$

Table C.4: Context bundle fields (schematic)

Group	Symbol	Representative fields
Session	s_{sess}	Session identifier, start timestamp, timer snapshot, local environment flags.
MBC summary	m	Instrument totals and bands (PHQ–9, GAD–7, PCL–5, WHODAS, composite risk).
Patient (anon.)	p_{anon}	Age band, relevant comorbidities, medication flags, encoded without direct identifiers.
Flow state	f	Current flow node, path history, pending tasks or deferred items.
Audit (anon.)	a_{anon}	Risk flags, safety checks performed, local log counters, redacted location/time hints.

The de-identification operator D enforces

$$D : \mathcal{C} \to \mathcal{C}_{\text{safe}} \quad \text{such that} \quad \forall c \in \mathcal{C} : \text{Identifiable}(D(c)) = \text{false}.$$

C.5 Layered Architectural Summary

At a higher level, the system can be interpreted as four composable layers:

$$\mathcal{L}_1 \to \mathcal{L}_2 \to \mathcal{L}_3 \to \mathcal{L}_4,$$

where

- $\mathcal{L}_1$ is the interaction layer (UI + flow engine).
- $\mathcal{L}_2$ is the clinical computation layer (MBC + timer).
- $\mathcal{L}_3$ is the semantic layer (context + prompt).
- $\mathcal{L}_4$ is the AI interface layer (LLM, post-processing).

Each layer exposes a map $\Gamma_i : \mathcal{L}_i \to \mathcal{L}_{i+1}$, preserving determinism and privacy.

Table C.5: Layered architecture, inputs and outputs

Layer	Representative components	Input	Output / invariant
$\mathcal{L}_1$ Interaction	UI shell, flow nodes, event handlers.	User actions, local configuration.	Deterministic flow state trajectory in $\mathcal{F}$.
$\mathcal{L}_2$ Clinical computation	Autoscore calculators, composite risk, timer engine.	Flow state, questionnaire items, clock.	MBC summary $\mathcal{M}$ and timer state $\mathcal{T}$.
$\mathcal{L}_3$ Semantic context	Context builder, de-identification, prompt assembler.	$(\mathcal{F}, \mathcal{M}, \mathcal{T}, \mathcal{P}, \mathcal{A})$.	De-identified context bundle and structured prompt in $\mathcal{R}$.
$\mathcal{L}_4$ AI interface	LLM client, streaming handler, post-processor.	Structured prompt.	Textual response with clinical recommendations and summaries.

C.6 Summary

The mathematical formulation in this appendix replaces diagrammatic figures with explicit state spaces, mappings, and invariants. Together, the definitions and tables show that SynapseCore can be viewed as:

- a deterministic labelled transition system for flow control,
- a collection of pure scoring and aggregation functions for MBC,
- a structured context-construction pipeline with strict de-identification,

- and a layered architecture in which each map has a clearly defined domain, codomain, and responsibility.

This formal perspective is intended to support rigorous reasoning about correctness, privacy, and extensibility across the entire system.

Appendix D: Configuration and Deployment

This appendix documents the configuration surface and deployment patterns for SynapseCore as a front end only, browser-based digital psychiatry workbench. The emphasis is on:

- Explicit environment variable conventions (“`.env`” files).
- Provider key handling for AI services.
- Theming and branding overrides that do not alter core logic.
- Deployment recipes for single-clinician, small-team, and institutional scenarios.

Throughout, the design goal is to keep configuration:

1. *Deterministic*: the same configuration yields the same behaviour across machines.
2. *Minimal*: only strictly necessary variables are exposed.
3. *Auditable*: every configuration parameter has a clear effect and a documented default.

D.1 Configuration Model and Environments

SynapseCore assumes a standard “12-factor” style environment model. Configuration is externalised via key-value pairs, loaded at build time or application start. We denote the configuration space by

$$\mathcal{K} = \{(k, v) \mid k \in \mathcal{N}, v \in \mathcal{V}\},$$

where $\mathcal{N}$ is the set of variable names and $\mathcal{V}$ the set of permissible values (strings, booleans, or numeric literals).
In practice, configuration values are typically supplied via one or more files of the form:

- `.env` (base defaults, checked into version control).
- `.env.local` (developer-specific overrides, not committed).
- `.env.production` (deployment-time overrides).

Let $\mathcal{K}_{\text{base}}$ be the set of pairs from `.env`, and $\mathcal{K}_{\text{site}}$ the site or machine specific overrides from `.env.local` or environment variables. Then the effective configuration $\mathcal{K}_{\text{eff}}$ is given by the shadowing rule

$$\mathcal{K}_{\text{eff}} = \mathcal{K}_{\text{base}} \cup \mathcal{K}_{\text{site}},$$

with the convention that if a key appears in both sets, the value in $\mathcal{K}_{\text{site}}$ takes precedence.

Table D.2: Examples of provider-specific configuration

OpenAI **Environment keys:** `VITE_OPENAI_API_KEY`, `VITE_OPENAI_BASE_URL`. **Notes:** Standard HTTP endpoint; supports model version pinning.

Anthropic **Environment keys:** `VITE_ANTHROPIC_API_KEY`, `VITE_ANTHROPIC_BASE_URL`. **Notes:** Used when `VITE_AI_PROVIDER=anthropic`.

Local model gateway **Environment keys:** `VITE_LOCAL_LLM_URL`. **Notes:** Proxy exposing an OpenAI-compatible API over a local network.

Key rotation. In institutional settings, keys should be rotated regularly. Let $k_p^{(t)}$ denote the key for provider p at rotation step t. The system assumes that for eac
A typical developer `.env.local` might contain:

```
VITE_APP_ENV=local
VITE_APP_BASE_URL=/
VITE_AI_PROVIDER=openai
VITE_LLM_MODEL_DEFAULT=gpt-4.1-mini
VITE_TELEMETRY_ENABLED=false
VITE_LOG_LEVEL=debug
```

The production configuration would tighten these values, as discussed in Section D.4.

D.2 Provider Keys and AI Configuration

SynapseCore treats AI providers as pluggable back ends. We denote the set of supported providers by

$$\mathcal{P}_{\text{AI}} = \{\text{openai}, \text{anthropic}, \text{local}, \dots\}.$$

For each provider $p \in \mathcal{P}_{\text{AI}}$ we assume a key-value pair

$$k_p = (\text{name}_p, \text{secret}_p),$$

with the invariant that no secret value is committed to version control. Instead, keys are injected via deployment-specific configuration.
A minimal set of environment variables for AI connectivity is:

```
VITE_AI_PROVIDER=openai

VITE_OPENAI_API_KEY=...
VITE_OPENAI_BASE_URL=https://api.openai.com/v1

VITE_ANTHROPIC_API_KEY=...
VITE_ANTHROPIC_BASE_URL=https://api.anthropic.com/v1

VITE_LOCAL_LLM_URL=http://localhost:11434/v1/chat/completions
```

The AI orchestration layer implements a dispatch map

$$\Omega : \mathcal{P}_{\text{AI}} \times \mathcal{R}_{\text{prompt}} \to \mathcal{R}_{\text{response}},$$

where $\mathcal{R}_{\text{prompt}}$ and $\mathcal{R}_{\text{response}}$ are the spaces of structured prompts and responses. The environment variable `VITE_AI_PROVIDER` selects which branch of Ω is activated at runtime.

Table D.2: Examples of provider-specific configuration
OpenAI **Environment keys:** `VITE_OPENAI_API_KEY`, `VITE_OPENAI_BASE_URL`. **Notes:** Standard HTTP endpoint; supports model version pinning.
Anthropic **Environment keys:** `VITE_ANTHROPIC_API_KEY`, `VITE_ANTHROPIC_BASE_URL`. **Notes:** Used when `VITE_AI_PROVIDER=anthropic`.
Local model gateway **Environment keys:** `VITE_LOCAL_LLM_URL`. **Notes:** Proxy exposing an OpenAI-compatible API over a local network.
Key rotation. In institutional settings, keys should be rotated regularly. Let $k_p^{(t)}$ denote the key for provider p at rotation step t. The system assumes that for each p there is a sequence

$$(k_p^{(0)}, k_p^{(1)}, k_p^{(2)}, \dots)$$

and that configuration updates occur atomically between sessions; the front end simply reads the currently active value for k_p.
Privacy guardrails. Configuration should enforce the principle that no raw identifiers are transmitted to AI providers. While the de-identification logic is implemented in code (see Appendix C), configuration can harden this behaviour:

```
VITE_AI_ALLOW_RAW_PHI=false
VITE_AI_MAX_TOKENS=2048
VITE_AI_TEMPERATURE_DEFAULT=0.1
VITE_AI_TOP_P_DEFAULT=0.9
```

Setting `VITE_AI_ALLOW_RAW_PHI=false` instructs the AI layer to abort or warn if it detects an attempt to send non-de-identified content.

D.3 Theming and Visual Overrides

SynapseCore separates clinical logic from visual presentation via a small set of design tokens. Let Θ denote the theme parameter space:

$$\Theta = \{\theta_{\text{palette}}, \theta_{\text{typography}}, \theta_{\text{spacing}}, \theta_{\text{border}}, \theta_{\text{motion}}\}.$$

A theme instance $\theta \in \Theta$ is realised as a collection of CSS variables, for example in a file like `src/styles/theme-clinic.css`. Environment variables are used to select which theme file is loaded at startup:

```
VITE_THEME_PRESET=clinic   # or: light, dark, high-contrast
```

Table D.3: Example design tokens for the "clinic" theme

Token	Example value	Role
`-sc-color-bg`	`#f8fafc`	Neutral background for main panels.
`-sc-color-surface`	`#ffffff`	Card and dialog background.
`-sc-color-accent`	`#2563eb`	Primary accent (buttons, links).
`-sc-color-danger`	`#b91c1c`	Risk or safety-critical highlights.
`-sc-font-sans`	`"Inter", system-ui`	Primary text font family.
`-sc-radius-lg`	`0.75rem`	Rounded corner radius for cards.

The mapping from configuration to styles can be summarised as:

$$\Sigma_{\text{theme}} : \text{Preset name} \to \Theta,$$

where, for example, $\Sigma_{\text{theme}}(\texttt{clinic})$ returns a curated set of palette and spacing tokens adapted for a clinical setting (high contrast, low distraction, consistent with institutional branding).
For advanced deployments, a site-specific CSS file can be supplied without modifying core code. For instance, an institution might define a file `theme-hospital-x.css` and configure:

```
VITE_THEME_PRESET=hospital-x
VITE_THEME_CUSTOM_URL=/assets/themes/hospital-x.css
```

The front end checks for `VITE_THEME_CUSTOM_URL` at runtime; if present, it attempts to load the referenced stylesheet.

D.4 Deployment Scenarios

We distinguish three deployment archetypes:

1. **Single-clinician laptop deployment** (standalone, no network dependency for most functions).
2. **Small-team local network deployment** (shared hosting within a clinic or academic group).
3. **Institutional deployment behind a reverse proxy** (hospital or university infrastructure).

D.4.1 Single-clinician deployment

In this scenario the clinician runs SynapseCore locally on a laptop, typically using a static file server or the development server.
Key properties:

- Configuration is stored in `.env.local` and never synced to shared repositories.
- AI providers may be restricted to local models or disabled entirely.
- Telemetry is fully local and may be turned off.

Example configuration:

```
VITE_APP_ENV=local
VITE_APP_BASE_URL=/

VITE_AI_PROVIDER=local
VITE_LOCAL_LLM_URL=http://localhost:11434/v1/chat/completions

VITE_TELEMETRY_ENABLED=false
VITE_LOG_LEVEL=info
```

D.4.2 Small-team network deployment

For a small team, SynapseCore can be built into a static bundle and served from an internal web server (for example, Nginx or Apache) accessible only on a private network.
A typical build-and-serve process:

```
# 1. Build the static bundle
npm install
npm run build

# 2. Copy dist/ to a web root on the internal server
rsync -av dist/ clinic-web:/var/www/synapsecore/
```

The corresponding server block (simplified Nginx example) might be:

```
server {
  listen 443 ssl;
  server_name synapsecore.clinic.local;

  root /var/www/synapsecore;
  index index.html;

  location / {
    try_files $uri /index.html;
  }
}
```

Environment variables are injected by building with a dedicated `.env.production`:

```
VITE_APP_ENV=staging
VITE_APP_BASE_URL=/   # or /synapsecore/ if subpath

VITE_AI_PROVIDER=openai
VITE_OPENAI_API_KEY=... (set via CI secrets)
VITE_TELEMETRY_ENABLED=true
VITE_LOG_LEVEL=warn
```

D.4.3 Institutional deployment with reverse proxy

In an institutional context, SynapseCore is typically deployed behind:

- an authentication gateway (for example, SSO or VPN),
- a reverse proxy that terminates TLS,
- strict outbound firewall rules.

In this setting, configuration must reflect institutional policies:

```
VITE_APP_ENV=prod
VITE_APP_BASE_URL=/synapsecore/

VITE_AI_PROVIDER=openai
VITE_OPENAI_API_KEY=${SECRET_OPENAI_KEY_FROM_VAULT}
VITE_OPENAI_BASE_URL=https://internal-proxy/hardened/openai

VITE_TELEMETRY_ENABLED=true
VITE_LOG_LEVEL=error
VITE_AI_ALLOW_RAW_PHI=false
```

AI traffic is passed through an institutional proxy that enforces whitelisting, logging, and potential additional redaction before requests leave the local network.

Table D.4: Summary of deployment archetypes

Scenario	Typical host	Users	AI provider	Key concerns
Single-clinician	Laptop, local dev server.	1	Local or disabled.	Simplicity; offline capability.
Small team	Clinic intranet server.	2–20	Cloud or local.	Shared configuration; role separation.
Institutional	Hospital or university web stack.	20+	Cloud via proxy.	Policy compliance; centralised observability.

D.5 Logging, Telemetry, and Data Boundaries

While the core system is front end only, configuration still governs how local logs and optional telemetry are handled. Let

$$\Lambda = \{\text{error}, \text{warn}, \text{info}, \text{debug}\}$$

denote the log level lattice, with the usual ordering error $\leq$ warn $\leq$ info $\leq$ debug.
The environment variable `VITE_LOG_LEVEL` selects a threshold $\lambda^\star \in \Lambda$. A log event with level λ is emitted if and only if $\lambda \geq \lambda^\star$.
An additional boolean `VITE_TELEMETRY_ENABLED` controls whether non-identifying usage metrics (for example, which flows are most frequently used, or which scales are most commonly completed) are persisted locally.

```
VITE_LOG_LEVEL=warn
VITE_TELEMETRY_ENABLED=false
```

In institutional deployments, telemetry may be redirected to a self-hosted endpoint instead of the browser-only store:

```
VITE_TELEMETRY_ENABLED=true
VITE_TELEMETRY_ENDPOINT=/api/synapsecore/metrics
```

All telemetry payloads must be constructed from summarised metadata; no free-text notes or raw identifiers are included. This constraint is an architectural invariant independent of configuration, but configuration can enforce an additional fail-safe by disabling telemetry entirely.

Table D.5: Suggested configuration profiles

Profile	Key settings	Intended use

`local-dev`	`VITE_APP_ENV=local`
	`VITE_AI_PROVIDER=local`
	`VITE_LOG_LEVEL=debug`
	`VITE_TELEMETRY_ENABLED=false`
		Experimental development on a personal machine; no external AI calls.

`clinic-demo`	`VITE_APP_ENV=staging`
	`VITE_AI_PROVIDER=openai`
	`VITE_LOG_LEVEL=info`
	`VITE_THEME_PRESET=clinic`
		Demonstrations or teaching within a clinic or university setting.

`institution-prod`	`VITE_APP_ENV=prod`
	`VITE_AI_PROVIDER=openai` via proxy
	`VITE_AI_ALLOW_RAW_PHI=false`
	`VITE_TELEMETRY_ENDPOINT=/api/...`
		Operational use on institutional infrastructure with strict privacy and logging.

D.6 Recommended Configuration Profiles

For reproducibility, it is useful to define named configuration profiles. A profile is a function

$$\Pi : \{\text{local}, \text{clinic}, \text{institution}\} \to \mathcal{K}_{\text{eff}},$$

that maps a scenario label to a concrete key set.

These profiles can be realised as small shell wrappers or CI/CD workflows that set the appropriate environment variables before running the build step.

D.7 Summary

This appendix has reformulated configuration and deployment not as an implementation detail but as a structured, auditable interface between SynapseCore and its hosting environment. The key points are:

- All critical behaviour (AI providers, theming, logging, telemetry) is controlled by explicit environment variables.
- Provider keys are never stored in the codebase; configuration supports rotation and proxy-based hardening.
- Visual theming is isolated via design tokens and can be overridden without modifying clinical logic.
- Deployment archetypes (single clinician, clinic, institution) correspond to well-defined configuration profiles with different security and observability envelopes.

Taken together, these conventions support the broader goals of the project: reproducibility, privacy, and safe experimentation with AI-assisted clinical tooling in psychiatry.

Appendix E: Prompt Templates and JSON Schemas

This appendix provides a detailed specification of the prompt templates and JSON schemas that govern all interactions between SynapseCore and external or local language models. The objective is to make the AI layer fully transparent, auditable, and reproducible by treating prompts and responses as typed artefacts rather than opaque text.
We proceed as follows:

- Introduce a taxonomy of prompt families and design goals.
- Formalise a canonical prompt structure derived from the context bundle.
- Specify JSON schemas for requests, responses, and structured clinical outputs.
- Provide concrete psychiatry-focused templates for key modes (session summary, risk explanation, treatment formulation, patient-facing explanation, technical debug).
- Describe validation, versioning, and safety constraints for all AI exchanges, including logging and traceability.

Throughout, we assume that all prompts are built exclusively from the de-identified context bundle described in Appendix C, and that no free text is injected into prompts without passing through the same structured layer.

E.1 Prompt Taxonomy and Design Objectives

Let $\mathcal{C}$ denote the space of valid context bundles, and let $\mathcal{M}$ denote the set of *prompt modes*. A prompt mode identifies a clinical or technical task (for example, session summary or risk explanation).
A prompt template is a function

$$\Phi : \mathcal{C} \times \mathcal{M} \to \mathcal{P},$$

where $\mathcal{P}$ is the space of well-formed prompt strings. A particular instantiation can be written as

$$P = \Phi(c, m),$$

with $c \in \mathcal{C}$ and $m \in \mathcal{M}$.
In SynapseCore we distinguish the following top-level families:

$$\mathcal{M} = \{m_{\text{summary}}, m_{\text{risk}}, m_{\text{plan}}, m_{\text{patient}}, m_{\text{debug}}\},$$

with the following interpretations:

- m_{summary} (`"session_summary"`): structured summaries for other clinicians.
- m_{risk} (`"risk_explanation"`): explanatory narratives that justify an existing risk band.
- m_{plan} (`"treatment_plan"`): draft treatment formulations and monitoring suggestions.
- m_{patient} (`"patient_letter"`): patient-facing explanations in plain language.
- m_{debug} (`"developer_debug"`): technical descriptions of internal state for developers.

Design objectives for all prompt families are:

1. **Deterministic structure.** For a fixed mode m, the overall skeleton of the prompt (section headings, ordering of elements) is invariant across contexts.
2. **Minimal data exposure.** Only those context fields that are required for the task are rendered into the prompt.

Table E.1: Core prompt mode taxonomy

Mode identifier	Intended reader	Primary objective
`"session_summary"`	Psychiatrist or trainee	Summarise the visit using MBC and structured flows.
`"risk_explanation"`	Psychiatrist or supervisor	Explain why a risk band is plausible and what it implies.
`"treatment_plan"`	Psychiatrist or team conference	Outline candidate treatment components and monitoring.
`"patient_letter"`	Patient or caregiver (via clinician)	Provide clear, non-stigmatising psychoeducation.
`"developer_debug"`	Developer or power user	Expose flow state, timers, and context mapping for debugging.

3. **De-identified content.** All direct identifiers (names, addresses, free text with identifying detail) are removed or replaced by generic tokens at context construction time.
4. **Clinical transparency.** The prompts can be inspected and read as if they were part of a structured consultation template.

E.2 Canonical Prompt Structure

Each prompt is represented as an ordered sequence of labelled sections. Formally, for a given mode m we define an index set

$$I(m) = \{0, 1, \ldots, L_m\},$$

and a family of section generators

$$S_m : \mathcal{C} \times I(m) \to \text{Strings},$$

so that the full prompt is given by

$$\Phi(c, m) = \text{join}\big(S_m(c, 0), S_m(c, 1), \ldots, S_m(c, L_m)\big).$$

For the session summary mode m_{summary}, we typically use sections of the form:

Table E.2: Example section catalogue for `"session_summary"`

Section label	Source in context bundle	Mandatory
`# ROLE AND TASK`	Static template for mode	Yes.
`# SESSION CONTEXT`	`session.id`, `timer.elapsedMs`, visit type	Yes.
`# MBC SCALES`	`mbc.phq9`, `mbc.gad7`, `mbc.pcl5`, `mbc.whodas`	Yes (if scales exist, otherwise marked missing).
`# RISK AND SAFETY`	`risk.acute`, `risk.chronic`, `risk.protective`	Yes.
`# PSYCHOSOCIAL CONTEXT`	`patient.history`, `patient.stressors`, supports	Optional, may be summarised.
`# CLINICAL FORMULATION (OPTIONAL)`	`note.formulationDraft`	Included only if clinician provided formulation text.

A typical high-level stencil for m_{summary} is:

```
# ROLE AND TASK
You are a psychiatrist assisting with a structured, measurement-based
session review. Summarize the session for another psychiatrist.

# SESSION CONTEXT
- Session ID: {{session.id}}
- Time elapsed: {{session.elapsed_minutes}} minutes
- Visit type: {{session.visit_type}}
```

```
- Setting: {{session.setting}}

# MBC SCALES
- PHQ-9: total={{mbc.phq9.total}}, band={{mbc.phq9.severity}}
- GAD-7: total={{mbc.gad7.total}}, band={{mbc.gad7.severity}}
- PCL-5: total={{mbc.pcl5.total}}, band={{mbc.pcl5.severity}}
- WHODAS: total={{mbc.whodas.total}}, band={{mbc.whodas.severity}}
- Composite risk index: {{mbc.composite_risk_band}}

# RISK AND SAFETY
- Acute risk flags: {{risk.acute_flags}}
- Chronic risk factors: {{risk.chronic_factors}}
- Protective factors: {{risk.protective_factors}}

# PSYCHOSOCIAL CONTEXT
{{text.psychosocial_context}}

# FUNCTIONAL STATUS
{{text.functional_status}}

# REQUESTED OUTPUT FORMAT
Write three sections with numbered headings:
1. Symptom profile and MBC interpretation
2. Risk and safety summary
3. Functional impact and suggested monitoring focus
```

Here the `{{...}}` placeholders represent deterministic substitution from the context bundle. The joining operator join inserts line breaks and delimiter markers consistently across all modes.

E.3 JSON Schemas for Requests and Responses

The SynapseCore front end does not work with raw prompt strings in isolation. Instead it uses structured JSON envelopes for all requests and responses at the AI boundary.

E.3.1 Request envelope

Let $\mathcal{J}_{\text{req}}$ be the set of valid request objects. Each $r \in \mathcal{J}_{\text{req}}$ corresponds to a tuple

$$r = (\text{provider}, \text{model}, \text{mode}, \text{prompt}, \theta, \mu),$$

where θ collects sampling parameters (for example, maximum tokens, temperature) and μ collects metadata.
A canonical request has the following JSON shape:

```
{
  "provider": "openai",
  "model": "gpt-4.1-mini",
  "mode": "session_summary",
  "prompt": "... assembled prompt string ...",
  "max_tokens": 1024,
  "temperature": 0.1,
  "top_p": 0.9,
  "metadata": {
    "session_id_hash": "hash_of_internal_id",
    "schema_version": "E.1",
    "deidentified": true,
    "ui_client": "synapsecore-web@1.0.0"
  }
}
```

Table E.3: AI request envelope fields

Field	Type	Description
`provider`	String	Logical provider key (for example, `"openai"`, `"local"`).
`model`	String	Model identifier recognised by the provider.
`mode`	String enum	Prompt mode, for example `"session_summary"`.
`prompt`	String	Final concatenated prompt generated by $\Phi(c, m)$.
`max_tokens`	Integer	Upper bound on completion tokens.
`temperature`	Number	Sampling temperature; typically low for clinical outputs.
`top_p`	Number	Nucleus sampling parameter.
`metadata.deidentified`	Boolean	Must be `true` before any request is sent.
`metadata.schema_version`	String	Version identifier for the prompt/schema pair.

The invariants for $r \in \mathcal{J}_{\text{req}}$ are:

1. All required fields (`provider`, `model`, `mode`, `prompt`) are present.
2. `metadata.deidentified` is explicitly `true`.
3. The `prompt` field is the image of some (c, m) under Φ, that is, no ad hoc free text is appended downstream.

E.3.2 Response envelope

Let $\mathcal{J}_{\text{res}}$ be the set of response objects. Each $s \in \mathcal{J}_{\text{res}}$ is represented as

$$s = (\text{text}, u, \mu'),$$

where u encodes usage information and μ' captures response metadata.
A typical response JSON object is:

```
{
  "text": "... model response text ...",
  "usage": {
    "prompt_tokens": 712,
    "completion_tokens": 228,
    "total_tokens": 940
  },
  "metadata": {
    "provider": "openai",
    "model": "gpt-4.1-mini",
    "received_at": "2025-11-27T17:14:03Z",
    "mode": "session_summary",
    "schema_version": "E.1"
  }
}
```

The front end then passes `text` through a mode-dependent post-processor:

$$\Psi_m : \mathcal{J}_{\text{res}} \to \mathcal{O}_m,$$

where $\mathcal{O}_m$ is the space of accepted outputs for mode m (for example, plain text or JSON objects matching a schema). If structured output is requested, Ψ_m attempts to parse the response as JSON and validate it against a mode-specific schema $\mathcal{S}_m$ (see Section E.5).

E.4 Clinical Prompt Templates

We now outline representative templates for core psychiatry tasks. In practice, each template is implemented as a pure function from the context bundle and mode to a string, but we present them as annotated stencils here.

E.4.1 Session summary ("session_summary")

Task. Produce a structured summary for another psychiatrist, grounded in MBC and structured flows.

```
# ROLE AND TASK
You are a psychiatrist assisting with a structured, measurement-based
session review for another psychiatrist. Summarise the consultation
based ONLY on the de-identified, structured information below.

# IMPORTANT CONSTRAINTS
- Do NOT invent data or diagnoses.
- If information is missing, explicitly say that it is not available.
- Do NOT contradict scale scores or existing risk bands.

# SESSION CONTEXT
- Session ID: {{session.id}}
- Time elapsed: {{session.elapsed_minutes}} minutes
- Visit type: {{session.visit_type}}
- Setting: {{session.setting}}
- Reason for visit (brief): {{text.reason_for_visit}}

# MEASUREMENT-BASED CARE (MBC) SCALES
- PHQ-9: total={{mbc.phq9.total}}, band={{mbc.phq9.severity}}
- GAD-7: total={{mbc.gad7.total}}, band={{mbc.gad7.severity}}
- PCL-5: total={{mbc.pcl5.total}}, band={{mbc.pcl5.severity}}
- WHODAS: total={{mbc.whodas.total}}, band={{mbc.whodas.severity}}
- Composite risk index: {{mbc.composite_risk_band}}

# RISK AND SAFETY
- Acute risk flags: {{risk.acute_flags}}
- Chronic risk factors: {{risk.chronic_factors}}
- Protective factors: {{risk.protective_factors}}
- System risk band (do NOT override): {{risk.current_band}}

# PSYCHOSOCIAL CONTEXT (DE-IDENTIFIED)
{{text.psychosocial_context}}

# FUNCTIONAL STATUS
{{text.functional_status}}

# CLINICIAN NOTES (OPTIONAL, DE-IDENTIFIED)
{{text.clinician_free_text}}

# REQUESTED OUTPUT FORMAT
Write in three numbered sections, using short paragraphs:

1. Current symptom profile and interpretation of MBC scores
2. Risk, safety considerations, and protective factors
3. Functional impact, psychosocial context, and monitoring priorities

Keep the tone neutral, professional, and suitable for inclusion in a
brief handover note to another psychiatrist.
```

E.4.2 Risk explanation ("risk_explanation")

Task. Explain and contextualise an existing risk band without reclassifying it.

```
# ROLE AND TASK
You are a psychiatrist asked to EXPLAIN the reasoning behind an
existing risk classification. You MUST NOT reclassify the risk level.
Instead, interpret the current risk band in light of the structured
data below.

# MBC SCALES
- PHQ-9: total={{mbc.phq9.total}}, band={{mbc.phq9.severity}}
- GAD-7: total={{mbc.gad7.total}}, band={{mbc.gad7.severity}}
- PCL-5: total={{mbc.pcl5.total}}, band={{mbc.pcl5.severity}}
- WHODAS: total={{mbc.whodas.total}}, band={{mbc.whodas.severity}}

# RISK SIGNALS
- System risk band (fixed): {{risk.current_band}}
- Acute risk flags: {{risk.acute_flags}}
- Chronic risk factors: {{risk.chronic_factors}}
- Protective factors: {{risk.protective_factors}}
- Recent changes since last visit: {{text.recent_change_summary}}

# REQUESTED OUTPUT FORMAT
Write three short sections:

1. Factors that INCREASE risk in this case
2. Factors that DECREASE or buffer risk (protective factors)
3. Interpretation of the current risk band and suggested level of
   clinical vigilance (for example, routine, enhanced, intensive),
   without issuing treatment orders.

Be explicit about uncertainties and missing information.
```

E.4.3 Treatment formulation ("treatment_plan")

Task. Provide a draft treatment formulation that the clinician can review, not a prescriptive order set.

```
# ROLE AND TASK
You are a psychiatrist generating a DRAFT treatment formulation based on
the structured information below. The human psychiatrist remains fully
responsible for all diagnostic and treatment decisions.

# INPUT DATA
- Symptom scales and bands (PHQ-9, GAD-7, PCL-5, WHODAS, composite risk).
- Key stressors and psychosocial context.
- Existing diagnoses (if any): {{text.existing_diagnoses}}
- Current medications (if any): {{text.current_medications}}
- Previous treatment response: {{text.previous_response}}

# REQUESTED OUTPUT FORMAT
Provide a concise treatment formulation with five sections:

1. Provisional diagnostic impressions (narrative, emphasising uncertainty)
2. Symptom-focused treatment targets (bullet list)
3. Psychotherapy recommendations (for example, CBT, trauma-focused work),
   with brief rationale
4. Medication considerations (if applicable), described at the level of
   classes or mechanisms, NOT specific doses
```

```
5. Monitoring and follow-up suggestions (for example, revisit MBC scales,
   frequency of follow-up)

Use conditional language ("could be considered", "may be appropriate").
Do NOT provide specific dosing schedules or detailed drug names.
```

E.4.4 Patient-facing explanation (`"patient_letter"`)

Task. Generate a patient-facing explanation in JSON form so that each section can be rendered separately in the UI.

```
# ROLE AND TASK
You are helping to draft a short explanation that a psychiatrist could
share with a patient. The goal is to explain the main findings and next
steps in simple, respectful language.

# INPUT DATA (DE-IDENTIFIED SUMMARY)
- Main symptom themes: {{text.symptom_themes}}
- Relevant scale bands (for example, "moderate depression", "mild anxiety")
- Overall risk band (for example, "low", "moderate"): {{risk.current_band}}
- Key strengths and supports: {{text.strengths_and_supports}}

# JSON OUTPUT SPECIFICATION
Return ONLY a single JSON object with the following fields:

{
  "reading_level": "very short description of approximate level",
  "summary_paragraph": "2-4 sentences in plain language",
  "symptoms_section": "how current symptoms are described to the patient",
  "strengths_section": "emphasis on strengths and coping",
  "next_steps_section": "what will likely happen next"
}

Constraints:
- Do NOT include any fields other than the five listed above.
- Do NOT include any patient identifiers (no names, dates of birth, or
  addresses).
- Use neutral, non-stigmatising language.
- Avoid technical jargon where possible; if you must use a term, briefly
  explain it.
```

E.4.5 Developer debug (`"developer_debug"`)

Task. Describe internal state for debugging and inspection.

```
# ROLE AND TASK
You are assisting a developer in understanding the current internal
state of the SynapseCore workbench. Use the structured data below to
describe what the system is doing, in technical but clear language.

# FLOW STATE
- Current node: {{flow.current_node}}
- Visited nodes (sequence): {{flow.visited_nodes}}
- Pending branches: {{flow.pending_branches}}

# TIMER STATE
- Running: {{timer.running}}
- Elapsed milliseconds (snapshot): {{timer.elapsedMs}}
```

```
# MBC SNAPSHOT
- PHQ-9: {{mbc.phq9.total}} ({{mbc.phq9.severity}})
- GAD-7: {{mbc.gad7.total}} ({{mbc.gad7.severity}})
- PCL-5: {{mbc.pcl5.total}} ({{mbc.pcl5.severity}})
- WHODAS: {{mbc.whodas.total}} ({{mbc.whodas.severity}})

# REQUESTED OUTPUT FORMAT
Produce:
1. A brief narrative description of the current flow position and how
   the user likely reached it.
2. A description of how the timer state will be interpreted by the UI.
3. A short list of potential edge cases a developer should test next
   (for example, branch conditions, timer pauses, missing scale data).
```

E.5 JSON Schemas for Structured Clinical Output

For modes that request structured output, SynapseCore defines explicit JSON schemas. Let $\mathcal{O}_m$ denote the space of valid outputs for mode m, and let $\mathcal{S}_m$ denote the corresponding JSON schema. A validator $V_m : \mathcal{O}_m \to \{0, 1\}$ is defined such that

$$V_m(o) = 1 \iff o \text{ satisfies } \mathcal{S}_m.$$

E.5.1 Session summary schema

When a session summary is requested in structured form, the following schema is used:

```
{
  "type": "object",
  "required": [
    "headline",
    "symptom_summary",
    "risk_summary",
    "functioning_summary",
    "monitoring_suggestions"
  ],
  "properties": {
    "headline": { "type": "string" },
    "symptom_summary": { "type": "string" },
    "risk_summary": { "type": "string" },
    "functioning_summary": { "type": "string" },
    "monitoring_suggestions": {
      "type": "array",
      "items": { "type": "string" }
    }
  },
  "additionalProperties": false
}
```

E.5.2 Risk explanation schema

For risk explanations with structured output:

```
{
  "type": "object",
  "required": [
    "risk_band",
```

```
    "key_contributors",
    "protective_factors",
    "clinical_caveats"
  ],
  "properties": {
    "risk_band": { "type": "string" },
    "key_contributors": {
      "type": "array",
      "items": { "type": "string" }
    },
    "protective_factors": {
      "type": "array",
      "items": { "type": "string" }
    },
    "clinical_caveats": { "type": "string" }
  },
  "additionalProperties": false
}
```

E.5.3 Treatment plan schema

For treatment formulation:

```
{
  "type": "object",
  "required": [
    "diagnostic_impressions",
    "targets",
    "psychotherapy",
    "medication_considerations",
    "monitoring"
  ],
  "properties": {
    "diagnostic_impressions": { "type": "string" },
    "targets": {
      "type": "array",
      "items": { "type": "string" }
    },
    "psychotherapy": {
      "type": "array",
      "items": { "type": "string" }
    },
    "medication_considerations": {
      "type": "array",
      "items": { "type": "string" }
    },
    "monitoring": {
      "type": "array",
      "items": { "type": "string" }
    }
  },
  "additionalProperties": false
}
```

E.5.4 Patient-facing explanation schema

For patient-facing explanations (repeated here for completeness):

```
{
  "type": "object",
  "required": [
    "reading_level",
    "summary_paragraph",
    "symptoms_section",
    "strengths_section",
    "next_steps_section"
  ],
  "properties": {
    "reading_level": { "type": "string" },
    "summary_paragraph": { "type": "string" },
    "symptoms_section": { "type": "string" },
    "strengths_section": { "type": "string" },
    "next_steps_section": { "type": "string" }
  },
  "additionalProperties": false
}
```

Table E.4: Mapping from modes to JSON schemas

Mode	Schema identifier	Structured fields
`"session_summary"`	`"SummarySchema_E1"`	Headline, symptom, risk, functioning, monitoring suggestions.
`"risk_explanation"`	`"RiskSchema_E1"`	Risk band, contributors, protective factors, caveats.
`"treatment_plan"`	`"PlanSchema_E1"`	Impressions, targets, psychotherapy, medication, monitoring.
`"patient_letter"`	`"PatientLetterSchema_E1"`	Reading level, summary, symptoms, strengths, next steps.
`"developer_debug"`	Plain text only	No structured schema in baseline build.

E.6 Validation, Versioning, and Safety Constraints

To ensure robustness and clinical safety, SynapseCore enforces a validation and versioning pipeline over all AI interactions.

E.6.1 Validation pipeline

Given a mode $m \in \mathcal{M}$, let $\mathcal{S}_m$ be the JSON schema (if any), and let V_m be its validator. Define

$$\Gamma_m : \mathcal{J}_{\text{res}} \to \{\text{accept}, \text{reject}\}$$

by

$$\Gamma_m(s) = \begin{cases} \text{accept}, & \text{if } m \text{ expects plain text only,} \\ \text{accept}, & \text{if } m \text{ expects JSON and } V_m(o) = 1 \text{ for parsed } o, \\ \text{reject}, & \text{otherwise.} \end{cases}$$

Informally, for structured modes:

1. Attempt to parse `text` as JSON.
2. Check `additionalProperties` and required fields.
3. If validation fails, discard or quarantine the response and surface a non-fatal error to the clinician.

E.6.2 Versioning

Each pair of prompt template and schema is assigned a version string, for example `"E.1"` or `"E.2"`. Let

$$\nu : \mathcal{M} \to \text{Strings}$$

map each mode to its current schema version. Then the metadata field `schema_version` in both request and response envelopes is set to $\nu(m)$.
When a template or schema is changed in a non-trivial way, $\nu(m)$ is incremented. Historical logs can then reconstruct which template and schema were in force for any given interaction, supporting:

- Retrospective analysis of behaviour across versions.
- Controlled roll-out of updated templates.
- Regression testing of validation rules.

E.6.3 Safety constraints

In addition to structural validation, several safety constraints are enforced:

- **No identifiers in prompts.** The context construction layer strips or replaces any direct identifiers before prompt assembly.
- **No identifiers in responses.** Responses may be scanned for patterns suggestive of names, addresses, or dates of birth. If such patterns are detected, the clinician is warned and the relevant segments may be redacted.
- **Mode-consistent behaviour.** Templates explicitly instruct models not to reclassify risk bands or override existing scores.
- **Length control.** Upper bounds on `max_tokens` reduce the risk of overly long, unstructured output.

E.7 Traceability, Logging, and Auditability

To support audit and research use, each AI interaction can be logged in a de-identified manner, subject to configuration (see Appendix 19.4).
Let $\mathcal{L}$ denote the space of log entries. For each request and response pair (r, s), a log entry $\ell \in \mathcal{L}$ may contain:

- Hashed session identifier.
- Timestamp and duration of the call.
- Mode, provider, and model.
- Schema version identifier.
- Token usage statistics.
- Validation outcome (success or failure).

Formally, we define a projection

$$\pi_{\log} : \mathcal{J}_{\text{req}} \times \mathcal{J}_{\text{res}} \to \mathcal{L}$$

that strips all textual content (prompts and completions) and retains only metadata and usage information.

Table E.5: Example log entry fields (de-identified)

Field	Type	Description
`session_id_hash`	String	Hashed or pseudonymised session identifier.
`mode`	String	Prompt mode used (for example, `"risk_explanation"`).
`provider` / `model`	String	Provider and model identifiers.
`schema_version`	String	Prompt/schema version at time of call.
`duration_ms`	Integer	Elapsed time between request and response.
`prompt_tokens` / `completion_tokens`	Integer	Token counts where available.
`validation_result`	String enum	Outcome of JSON and safety validation.

These logs can then be analysed to monitor:

- Frequency of each mode in routine use.
- Typical prompt and completion lengths.

- Validation failure rates across schema versions.
- Latency and resource usage per provider.

Crucially, no clinical free text or patient identifiers are persisted in these logs.

E.8 Summary

In this appendix we have made explicit the design of the prompt and JSON schema layer that connects SynapseCore to external or local language models. The key principles are:

- Prompts are generated by deterministic template functions $\Phi(c, m)$ from de-identified context bundles and mode labels.
- A small but clinically meaningful taxonomy of modes supports different tasks (summary, risk explanation, treatment planning, patient-facing explanation, developer debugging) without mode confusion.
- Structured outputs are governed by mode-specific JSON schemas $\mathcal{S}_m$ and validators V_m, so that only well-formed responses are accepted.
- Version tags allow prompt templates and schemas to evolve while preserving backward traceability.
- De-identification, safety constraints, and de-identified logging make the AI layer compatible with clinical, educational, and research use in digital psychiatry.

By treating prompt design as a first-class, mathematically specified component of the system, SynapseCore aims to reduce opacity and to make AI-assisted reasoning accessible to audit, critique, and improvement by clinicians, researchers, and developers alike.

Appendix F: Telemetry and Metrics Specification

This appendix specifies the telemetry model for SynapseCore, with a focus on:

- The OpenTelemetry (OTEL) span taxonomy used to represent user interaction, flow execution, and AI calls.
- Event types recorded as span events and log records, with explicit semantic meanings.
- Metrics (counters and histograms) used to quantify usage and performance.
- Privacy and configuration constraints that govern what may be recorded and exported (see also Appendix 19.4).

The primary design goal is to make telemetry:

1. *Clinically safe*: no patient identifiers or free text are exported.
2. *Architecturally transparent*: all spans and events have predictable names and attribute keys.
3. *Useful for optimisation*: metrics can be used to detect performance bottlenecks and workflow friction.
4. *Configurable*: telemetry can be disabled or routed to a local endpoint as required.

F.1 Telemetry Scope and Conceptual Model

We view telemetry as a mapping from concrete runtime activity to de-identified traces and metrics. Let $\mathcal{A}$ denote the set of runtime activities[7] and let $\mathcal{T}$ denote the space of trace elements (spans, events, metrics). The OTEL instrumentation defines a function

$$\Theta : \mathcal{A} \to \mathcal{T},$$

subject to the invariant that all derived elements in $\mathcal{T}$ are free of direct identifiers and do not contain arbitrary clinical free text.

Concretely, telemetry covers:

- Session lifecycle (opening, saving, closing).
- Flow traversal (nodes entered, branches taken, cancellations).
- Measurement based care (MBC) completion events.
- Timer behaviour (start, pause, resume, snapshot).
- AI requests and responses (latency and token usage only).
- Configuration relevant events (for example, telemetry toggles, theme changes).

Telemetry does *not* capture:

- Free text note content.
- Patient names, dates of birth, addresses, or similar.
- Raw AI prompts or completions.

[7] For example, starting a session, completing a scale, or sending an AI request.

Instead, all identifiers are reduced to stable hashes or coarse categories (for example, session duration bucket, flow id, or scale type).

F.2 OTEL Resource Model and Namespaces

All OTEL data emitted by SynapseCore share a common resource description, allowing downstream back ends to group and filter traces.

Table F.1: OTEL resource attributes

Attribute key	Example value	Meaning
`service.name`	`"synapsecore-web"`	Stable identifier for this front end.
`service.version`	`"1.0.0"`	Application build version.
`deployment.environment`	`"local"`, `"staging"`, `"prod"`	Matches `VITE_APP_ENV`.
`telemetry.sdk.name`	`"opentelemetry-js"`	OTEL SDK name.
`telemetry.sdk.language`	`"javascript"`	Language of the instrumented runtime.
`ui.theme`	`"clinic"`, `"dark"`	Theme preset in effect.

Span names and metrics are grouped into logical namespaces, all prefixed with `"synapsecore."`. For example:

- `synapsecore.session.*` for session lifecycle.
- `synapsecore.flow.*` for flow traversal.
- `synapsecore.mbc.*` for MBC related events.
- `synapsecore.ai.*` for AI calls and validation.
- `synapsecore.timer.*` for timing.

This naming convention supports consistent querying in observability back ends.

F.3 Span Taxonomy

Spans represent units of work, typically corresponding to user-visible actions or internal operations. Let $\mathcal{S}$ be the set of span kinds used in SynapseCore. Each span $s \in \mathcal{S}$ is identified by a name n_s and is associated with a set of attributes $A_s = \{(k_i, v_i)\}$.

F.3.1 Session spans

Session spans capture the lifecycle of a clinical session.

Table F.2: Session related spans

Span name	Kind	Key attributes
`"synapsecore.session.start"`	Internal	`session_id_hash`, `visit_type`, `start_source`.
`"synapsecore.session.save"`	Internal	`session_id_hash`, `save_reason` (auto, manual).
`"synapsecore.session.close"`	Internal	`session_id_hash`, `duration_ms_bucket`.

All session spans share the attribute `session_id_hash`, a stable pseudonym that never exposes the underlying identifier.

F.3.2 Flow spans

Flow spans capture traversal through the declarative flow graph.

Table F.3: Flow related spans

Span name	Kind	Key attributes
`"synapsecore.flow.run"`	Internal	`flow_id`, `entry_node`, `exit_node`.
`"synapsecore.flow.step"`	Internal	`flow_id`, `node_id`, `step_type` (questionnaire, note, summary).
`"synapsecore.flow.branch"`	Internal	`flow_id`, `from_node`, `to_node`, `condition_label`.

The span `"synapsecore.flow.run"` is started when the flow is entered and finished upon completion or cancellation. Individual `flow.step` spans are nested inside it.

F.3.3 MBC spans

MBC spans represent the computation of psychometric scores and derived indices.

Table F.4: MBC related spans

Span name	Kind	Key attributes
`"synapsecore.mbc.calculate"`	Internal	`scale_id` (PHQ-9, GAD-7, PCL-5, WHO-DAS), `item_count`.
`"synapsecore.mbc.composite"`	Internal	`composite_id`, `source_scales`.
`"synapsecore.mbc.band-map"`	Internal	`scale_id`, `band` (minimal, mild, etc.).

These spans do not include raw item responses; they only record scale identifiers, item counts, and resulting bands.

F.3.4 AI spans

AI spans capture calls to external or local language models.

Table F.5: AI related spans

Span name	Kind	Key attributes
`"synapsecore.ai.request"`	Client	`provider`, `model`, `mode`, `schema_version`.
`"synapsecore.ai.validate"`	Internal	`mode`, `schema_id`, `result` (success, failure).
`"synapsecore.ai.safety-check"`	Internal	`mode`, `safety_flag` (phi_suspected, length_exceeded).

The `"synapsecore.ai.request"` span wraps the entire HTTP call to the AI provider and records latency via the span duration. Token usage is recorded as span attributes or separate metrics (see Section F.5).

F.3.5 Timer spans

The timer engine generates spans when its state changes.

Table F.6: Timer related spans

Span name	Kind	Key attributes
`"synapsecore.timer.session"`	Internal	`session_id_hash`, `elapsed_ms_snapshot`.
`"synapsecore.timer.transition"`	Internal	`transition_type` (start, pause, resume, reset).

F.4 Events and Log Records

Span events represent discrete points of interest within a span. Log records provide an additional channel for summarised status information. Let $\mathcal{E}$ be the set of event types and $\mathcal{L}$ the set of log record types.

F.4.1 Span events

Span events are attached to spans via OTEL and carry a name and attributes. Examples of event types are given in Table F.7.

Table F.7: Span events and semantic meanings

Event name	Attached to	Semantic meaning
`"flow.node.entered"`	`synapsecore.flow.run`	Flow has entered a specific node; attributes: `flow_id`, `node_id`.
`"flow.node.exited"`	`synapsecore.flow.run`	Flow has completed a node; attributes include `node_id`, `duration_ms`.
`"mbc.scale.completed"`	`synapsecore.mbc.calculate`	User has completed a scale; attributes: `scale_id`, `band`.
`"risk.band.changed"`	`synapsecore.mbc.composite`	Composite risk band updated; attributes: `old_band`, `new_band`.
`"ai.response.received"`	`synapsecore.ai.request`	AI response arrived; attributes: `completion_tokens`.
`"ai.validation.failed"`	`synapsecore.ai.validate`	Structured output did not match schema; attributes: `error_code`.
`"timer.snapshot"`	`synapsecore.timer.session`	Timer snapshot taken; attributes: `elapsed_ms`.

These events allow downstream tools to reconstruct detailed execution sequences without storing clinical content.

F.4.2 Log records

Log records provide summarised messages, generally at the level of `info`, `warn`, or `error`. They are governed by `VITE_LOG_LEVEL` (see Appendix 19.4).
Representative log record types include:

- `"synapsecore.telemetry.init"` Emitted when telemetry is initialised; attributes: `exporter_type`, `sampling_ratio`.
- `"synapsecore.telemetry.disabled"` Emitted when telemetry is explicitly disabled by configuration.
- `"synapsecore.ai.export.error"` Emitted when an AI request cannot be exported to the telemetry back end (for example, network error).
- `"synapsecore.schema.mismatch"` Emitted when a payload uses an unexpected schema version.

No clinical data are included in log messages; they carry only technical identifiers and coarse status codes.

F.5 Metrics: Counters and Histograms

OpenTelemetry metrics complement spans and events by providing aggregated statistics. Let $\mathcal{M}_{\text{metrics}}$ be the set of metric instruments (counters, histograms). Each metric $m \in \mathcal{M}_{\text{metrics}}$ is identified by a name and has a well defined unit.

F.5.1 Session and flow metrics

Table F.8: Session and flow metrics

Metric name	Type / unit	Description
"synapsecore.session.count"	Counter (1)	Number of sessions opened; attributes: visit_type.
"synapsecore.session.duration"	Histogram (ms)	Distribution of session durations; attributes: visit_type.
"synapsecore.flow.step.duration"	Histogram (ms)	Time spent per flow node; attributes: flow_id, node_id.
"synapsecore.flow.branch.count"	Counter (1)	Count of branch transitions; attributes: flow_id, condition_label.

F.5.2 MBC metrics

Table F.9: MBC metrics

Metric name	Type / unit	Description
"synapsecore.mbc.scale.completed"	Counter (1)	Number of completed scales; attributes: scale_id, band.
"synapsecore.mbc.composite.band"	Counter (1)	Composite risk band counts; attributes: band.
"synapsecore.mbc.items.missing"	Counter (1)	Incomplete scale attempts; attributes: scale_id.

F.5.3 AI metrics

Table F.10: AI metrics

Metric name	Type / unit	Description
"synapsecore.ai.request.count"	Counter (1)	Total number of AI requests; attributes: provider, mode.
"synapsecore.ai.latency"	Histogram (ms)	Latency distribution; attributes: provider, mode.
"synapsecore.ai.prompt.tokens"	Histogram (tokens)	Prompt token counts; attributes: provider, mode.
"synapsecore.ai.completion.tokens"	Histogram (tokens)	Completion token counts; attributes: provider, mode.
"synapsecore.ai.validation.failures"	Counter (1)	Number of schema validation failures; attributes: mode, schema_id.

F.5.4 Timer and usability metrics

Table F.11: Timer and usability metrics

Metric name	Type / unit	Description
"synapsecore.timer.active.duration"	Histogram (ms)	Distribution of active timer durations per session.
"synapsecore.timer.pause.count"	Counter (1)	How often the session timer is paused.
"synapsecore.ui.error.count"	Counter (1)	Number of user-visible error messages; attributes: error_type.
"synapsecore.ui.undo.count"	Counter (1)	Number of undo operations, as a proxy for usability friction.

F.6 Sampling, Privacy, and Configuration

Telemetry is further constrained by configuration (see Appendix 19.4). Let σ denote a sampling ratio in $[0,1]$ configured via environment variables. For any activity $a \in \mathcal{A}$, the probability that a corresponding trace element is emitted is:

$$\Pr\big(\Theta(a) \text{ is recorded}\big) = \sigma.$$

Typical settings include:

- $\sigma = 0$ for completely offline, no-telemetry builds.
- $\sigma \approx 0.1$ for development or high volume testing.
- $\sigma \in [0.01, 0.1]$ for institutional deployments.

Privacy invariants:

1. No patient identifiers or free text are placed in span attributes, events, or metrics.
2. Session identifiers are reduced to a one way hash or pseudonym.
3. AI prompts and completions are never exported as telemetry payloads, only their lengths and validation statuses.
4. Telemetry can be entirely disabled via `VITE_TELEMETRY_ENABLED=false`.

Configuration parameters that influence telemetry include:

- `VITE_TELEMETRY_ENABLED` Boolean master switch for instrumentation export.
- `VITE_TELEMETRY_ENDPOINT` Local or institutional endpoint for OTEL export.
- `VITE_LOG_LEVEL` Threshold for emitting log records.
- Optional future keys, such as `VITE_TELEMETRY_SAMPLING_RATIO`, to control σ.

F.7 Summary

This appendix has specified the telemetry and metrics layer of SynapseCore in terms of:

- A coherent OTEL resource model and span namespace.
- A span taxonomy covering sessions, flows, MBC, AI calls, and timer events.
- A catalogue of span events and log records with clearly defined semantics.
- Metrics that quantify usage, latency, and potential friction points in clinical workflows.
- Sampling and privacy invariants ensuring that telemetry remains de-identified and under clinician or institutional control.

By treating telemetry as a structured, formally specified subsystem, SynapseCore enables observability and performance tuning without compromising privacy or turning the AI layer into a black box. All instrumentation described here is optional and can be disabled or redirected according to local policy.

References

Abdelwanis, A., et al. (2024). Exploring the risks of automation bias in healthcare artificial intelligence applications: A bowtie analysis [Check journal name, volume, issue, and page range in your library database; details may have been updated after early online publication]. *Journal of Nuclear Safety and Security Research*. https://doi.org/10.1016/j.jnlssr.2024.06.001

Ali, M., Ali, S., Abbas, Q., Abbas, Z., & Lee, S. W. (2025). Artificial intelligence for mental health: A narrative review of applications, challenges, and future directions in digital health. *Digital Health*, *11*, 20552076251395548. https://doi.org/10.1177/20552076251395548

Alobayli, N., Alshammari, N., et al. (2023). The impact of medical record documentation on physician stress, job satisfaction, and burnout: A systematic review. *Medical Journal of Cairo University*, *91*, 151–165.

American Psychiatric Association. (2021). Resource document on digital mental health 101 [Prepared by the APA Work Group on Telepsychiatry and Task Force on Telehealth; accessed 21 November 2025].

Appelbaum, P. S., & Grisso, T. (1988). Assessing patients' capacities to consent to treatment. *New England Journal of Medicine*, *319*(25), 1635–1638. https://doi.org/10.1056/NEJM198812223192504

Arndt, B. G., Beasley, J. W., Watkinson, M. D., Temte, J. L., Tuan, W.-J., Sinsky, C. A., & Gilchrist, V. J. (2017). Tethered to the EHR: Primary care physician workload assessment using EHR event log data and time-motion observations. *Annals of Family Medicine*, *15*(5), 419–426. https://doi.org/10.1370/afm.2121

Asgari, E., Ahmed, A., Ward, M., et al. (2024). Impact of electronic health record use on cognitive load and its correlation with physician burnout: A cross-sectional study. *BMC Medical Informatics and Decision Making*, *24*(1), 175. https://doi.org/10.1186/s12911-024-02345-y

Babor, T. F., McRee, B. G., Kassebaum, P. A., Grimaldi, P. L., Ahmed, K., & Bray, J. (2007). Screening, brief intervention, and referral to treatment (SBIRT): Toward a public health approach to the management of substance abuse. *Substance Abuse*, *28*(3), 7–30. https://doi.org/10.1300/J465v28n03_03

Belsher, B. E., Smolenski, D. J., Pruitt, L. D., Bush, N. E., Beech, E. H., Workman, D. E., Morgan, R. L., Evatt, D. P., Tucker, J., & Skopp, N. A. (2019). Prediction models for suicide attempts and deaths: A systematic review and simulation. *JAMA Psychiatry*, *76*(6), 642–651. https://doi.org/10.1001/jamapsychiatry.2019.0174

Bender, D., & Sartipi, K. (2013). HL7 FHIR: An agile and restful approach to healthcare information exchange. *2013 IEEE 26th International Symposium on Computer-Based Medical Systems (CBMS)*, 326–331. https://doi.org/10.1109/CBMS.2013.6627810

Bourgeois, J. A., et al. (2019). Decisional capacity determinations in consultation-liaison psychiatry: APA resource document [Available from the APA website].

Boxwala, A. A., Rocha, B. H., Maviglia, S., Kashyap, V., Meltzer, S., Kim, J., Gandhi, T., Kuperman, G., Turchin, A., & Bates, D. W. (2011). A multi-layered framework for disseminating knowledge for computer-based clinical decision support. *Journal of the American Medical Informatics Association*, *18*(Suppl 1), i132–i139. https://doi.org/10.1136/amiajnl-2010-000063

Busch, F., Hoffmann, L., Rueger, C., van Dijk, E. H. C., Kader, R., Ortiz-Prado, E., Makowski, M. R., Saba, L., Hadamitzky, M., Kather, J. N., Truhn, D., Cuocolo, R., Adams, L. C., & Bressem, K. K. (2025). Current applications and challenges in large language models for patient care: A systematic review. *Communications Medicine*, *5*(1), 26. https://doi.org/10.1038/s43856-024-00717-2

Bush, G., Fink, M., Petrides, G., Dowling, F., & Francis, A. (1996). Catatonia. I. rating scale and standardized examination. *Acta Psychiatrica Scandinavica*, *93*(2), 129–136. https://doi.org/10.1111/j.1600-0447.1996.tb09814.x

Bush, K., Kivlahan, D. R., McDonell, M. B., Fihn, S. D., & Bradley, K. A. (1998). The AUDIT alcohol consumption questions (AUDIT-C): An effective brief screening test for problem drinking. *Archives of Internal Medicine*, *158*(16), 1789–1795. https://doi.org/10.1001/archinte.158.16.1789

Campanozzi, L. L. (2023). The role of digital literacy in achieving health equity in the third millennium society: A literature review. *Frontiers in Public Health*, *11*, 1109323. https://doi.org/10.3389/fpubh.2023.1109323

Carlson, D. M., & Yarns, B. C. (2023). Managing medical and psychiatric multimorbidity in older patients. *Therapeutic Advances in Psychopharmacology*, *13*, 1–12. https://doi.org/10.1177/20451253231195274

Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M. A., Guo, J., Li, P., & Riddell, A. (2017). Stan: A probabilistic programming language. *Journal of Statistical Software*, *76*(1), 1–32. https://doi.org/10.18637/jss.v076.i01

Choudhary, S. V. (2025). The evolving role of digital navigators to enhance mental health equity. *Psychiatric Services*.

Cleeland, C. S., & Ryan, K. M. (1994). Pain assessment: Global use of the brief pain inventory. *Annals of the Academy of Medicine, Singapore*, *23*(2), 129–138.

Connolly, S. L., Miller, C. J., Kearney, L. K., & Bauer, M. S. (2021). A systematic review of telehealth and digital mental health technology use in mental health services. *Psychological Services*, *18*(4), 738–760. https://doi.org/10.1037/ser0000429

Cruz-Gonzalez, P., He, A. W. J., Lam, E. P. P., Ng, I. M. C., Li, M. W., Hou, R., Chan, J. N. M., Sahni, Y., Vinas Guasch, N., Miller, T., Lau, B. W. M., & Sánchez Vidaña, D. I. (2025). Artificial intelligence in mental health care: A systematic review of diagnosis, monitoring, and intervention applications. *Psychological Medicine*, *55*, e18. https://doi.org/10.1017/S0033291724003295

Dey, A. K., Lewis, Z., Posel, J., Pan, R. Y., & Wang, K. (2025). Quantifying care, qualifying experiences: A systematic review of measurement-based care in psychiatry from patient and provider perspectives. *BMJ Mental Health*, *28*(1), e301663. https://doi.org/10.1136/bmjment-2025-301663

Dixon, L., et al. (2003). Family psychoeducation as an evidence-based practice. *Psychiatric Services*, *54*(4), 495–501. https://doi.org/10.1176/appi.ps.54.4.495

Dolin, R. H., Mandel, J. C., Kreda, D. A., Mandl, K. D., Kohane, I. S., & Ramoni, R. B. (2016). SMART on FHIR: A standards-based, interoperable apps platform for electronic health records. *Journal of the American Medical Informatics Association*, *23*(5), 899–908. https://doi.org/10.1093/jamia/ocv189

Downing, N. L., Bates, D. W., & Longhurst, C. A. (2018). Physician burnout in the electronic health record era: Are we ignoring the real cause? *Annals of Internal Medicine*, *169*(1), 50–51. https://doi.org/10.7326/M18-0139

Dworkin, R. H., Turk, D. C., Farrar, J. T., Haythornthwaite, J. A., Jensen, M. P., Katz, N. P., et al. (2005). Core outcome measures for chronic pain clinical trials: IMMPACT recommendations. *Pain*, *113*(1–2), 9–19. https://doi.org/10.1016/j.pain.2004.09.012

Elgin, C. Y., & Elgin, C. (2024). Ethical implications of AI-driven clinical decision support systems on healthcare resource allocation: A qualitative study of healthcare professionals' perspectives. *BMC Medical Ethics*, *25*(148), 1–15. https://doi.org/10.1186/s12910-024-01151-8

European Parliament and Council of the European Union. (2016a). Regulation (eu) 2016/679 (general data protection regulation), article 4(5): Definition of pseudonymisation [Article 4(5) of the General Data Protection Regulation defining "pseudonymisation" as the processing of personal data in such a manner that they can no longer be attributed to a specific data subject without the use of additional information. Consolidated text available at https://gdpr-info.eu/art-4-gdpr/ (accessed 2025-11-25).].

European Parliament and Council of the European Union. (2016b). Regulation (eu) 2016/679 (general data protection regulation), recital 26: Not applicable to anonymous data [Recital 26 of the General Data Protection Regulation clarifying that data protection principles do not apply to anonymous information that does not relate to an identified or identifiable natural person. Consolidated text available at https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32016R0679 (accessed 2025-11-25).].

European Parliament and Council of the European Union. (2017). Regulation (EU) 2017/745 of the european parliament and of the council of 5 april 2017 on medical devices [Regulation (EU) 2017/745 on medical devices (Medical Device Regulation, MDR), replacing Directives 90/385/EEC and 93/42/EEC. Available at https://eur-lex.europa.eu/eli/reg/2017/745/oj (accessed 2025-11-25).].

Fassassi, S., Bianchi, Y., Stiefel, F., & Waeber, G. (2009). Assessment of the capacity to consent to treatment in patients admitted to acute medical wards. *BMC Medical Ethics*, *10*, 15. https://doi.org/10.1186/1472-6939-10-15

Fiorillo, A., et al. (2015). Efficacy of psychoeducational family intervention for bipolar i disorder: A controlled, prospective study. *Journal of Affective Disorders*, *172*, 291–297. https://doi.org/10.1016/j.jad.2014.10.020

Firth, J., Torous, J., Nicholas, J., Carney, R., Pratap, A., Rosenbaum, S., & Sarris, J. (2017). The efficacy of smartphone-based mental health interventions for depressive symptoms: A meta-analysis of randomized controlled trials. *World Psychiatry*, *16*(3), 287–298. https://doi.org/10.1002/wps.20472

Fiske, A., Henningsen, P., & Buyx, A. (2019). Your robot therapist will see you now: Ethical implications of embodied artificial intelligence in psychiatry, psychology, and psychotherapy. *Journal of Medical Internet Research*, *21*(5), e13216. https://doi.org/10.2196/13216

Fortney, J. C., Unützer, J., Wrenn, G., Pyne, J. M., Smith, G. R., Schoenbaum, M., & Harbin, H. T. (2017a). A tipping point for measurement-based care. *Psychiatric Services*, *68*(2), 179–188. https://doi.org/10.1176/appi.ps.201500439

Fortney, J. C., Unützer, J., Wrenn, G., Pyne, J. M., Smith, G. R., Schoenbaum, M., & Harbin, H. T. (2017b). A tipping point for measurement-based care. *Psychiatric Services*, *68*(2), 179–188. https://doi.org/10.1176/appi.ps.201500439

Gaber, F., Shaik, M., Allega, F., Bilecz, A. J., Busch, F., Goon, K., Franke, V., & Akalin, A. (2025). Evaluating large language model workflows in clinical decision support for triage and referral and diagnosis [Evaluation of LLM-based clinical workflows; please confirm author list, volume/issue and DOI.]. *NPJ Digital Medicine*, *8*, 263.

Goddard, K., Roudsari, A., & Wyatt, J. C. (2012). Automation bias: A systematic review of frequency, effect mediators, and mitigators. *Journal of the American Medical Informatics Association*, *19*(1), 121–127. https://doi.org/10.1136/amiajnl-2011-000089

Goodman, W. K., Price, L. H., Rasmussen, S. A., Mazure, C., Fleischmann, R. L., Hill, C. L., Heninger, G. R., & Charney, D. S. (1989). The Yale–Brown obsessive compulsive scale: I. development, use, and reliability. *Archives of General Psychiatry*, *46*(11), 1006–1011. https://doi.org/10.1001/archpsyc.1989.01810110048007

Guo, T., Xiang, Y.-T., Xiao, L., Hu, C.-Q., Chiu, H. F. K., Ungvari, G., Correll, C. U., Lai, K. Y. C., Feng, L., Geng, Y., Feng, Y., & Wang, G. (2015). Measurement-based care versus standard care for major depression: A randomized controlled trial with blind raters. *American Journal of Psychiatry*, *172*(10), 1004–1013. https://doi.org/10.1176/appi.ajp.2015.14050652

Hansen, M., & Hasselbring, W. (2025). Instrumentation of software systems with opentelemetry for software visualization. *Softwaretechnik-Trends*, *45*(1), 17–19.

Health and Social Care Board Northern Ireland. (2018). Working together: A pathway for children and young people through CAMHS.

Health Service Executive. (2018). Consultation-liaison psychiatry: A model of care for ireland [National model-of-care document].

HL7 International. (2019). HL7 FHIR release 4: FHIR R4 specification [Accessed 2025-11-25].

Holmgren, A. J. (2024). Electronic health record usability and clinician burnout: A systematic review. *BMC Medical Informatics and Decision Making*.

Holmgren, A. J., Apathy, N., Downing, N. L., Pfeifer, E., Posnack, S., & Ratwani, R. M. (2024). Electronic health record usability, satisfaction, and physician burnout among family physicians [Exact article number to be verified]. *JAMA Network Open*, *7*(1), e2345678.

Huckvale, K., Torous, J., & Larsen, M. E. (2019). Assessment of the data sharing and privacy practices of smartphone apps for depression and smoking cessation. *JAMA Network Open*, *2*(4), e192542. https://doi.org/10.1001/jamanetworkopen.2019.2542

Hussey, M. A., & Hughes, J. P. (2007). Design and analysis of stepped wedge cluster randomized trials. *Contemporary Clinical Trials*, *28*(2), 182–191. https://doi.org/10.1016/j.cct.2006.05.007

International Medical Device Regulators Forum. (2013). Software as a medical device (SaMD): Key definitions [Foundational document defining “Software as a Medical Device” (SaMD) and related terminology for global regulatory convergence. Available at https://www.imdrf.org/sites/default/files/docs/imdrf/final/technical/imdrf-tech-131209-samd-key-definitions-140901.pdf (accessed 2025-11-25).].

International Medical Device Regulators Forum. (2017). Software as a medical device (SaMD): Clinical evaluation [Guidance describing principles and evidence requirements for the clinical evaluation of SaMD, including clinical association, analytical validation, and clinical validation. Available at https://www.imdrf.org/sites/default/files/docs/imdrf/final/technical/imdrf-tech-170921-samd-clinical-evaluation_0.pdf (accessed 2025-11-25).].

Issenberg, S. B., McGaghie, W. C., Petrusa, E. R., Gordon, D. L., & Scalese, R. J. (2005). Features and uses of high-fidelity medical simulations that lead to effective learning: A BEME systematic review. *Medical Teacher*, *27*(1), 10–28. https://doi.org/10.1080/01421590500046924

Jones, C., Thornton, J., & Wyatt, J. C. (2023). Artificial intelligence and clinical decision support: Clinicians' perspectives on trust, trustworthiness, and liability. *Medical Law Review*, *31*(4), 501–531. https://doi.org/10.1093/medlaw/fwad013

Kane, N. B., et al. (2022). Difficult capacity cases—the experience of liaison psychiatrists. *BJPsych Bulletin*, *46*(4), 228–235. https://doi.org/10.1192/bjb.2021.54

Kilbourne, A. M., Beck, K., Spaeth-Rublee, B., Ramanuj, P., O'Brien, R. W., Tomoyasu, N., & Pincus, H. A. (2018). Measuring and improving the quality of mental health care: A global perspective. *World Psychiatry*, *17*(1), 30–38. https://doi.org/10.1002/wps.20482

Kilbourne, A. M., Goodrich, D. E., Miake-Lye, I., Braganza, M. Z., & Bowersox, N. W. (2019). Quality enhancement research initiative implementation roadmap: Learning health system lessons from the VA healthcare system. *Health Services Research*, *54*(Suppl 1), 243–248. https://doi.org/10.1111/1475-6773.13109

Kilminster, S. M., & Jolly, B. C. (2000). Effective supervision in clinical practice settings: A literature review. *Medical Education*, *34*(10), 827–840. https://doi.org/10.1046/j.1365-2923.2000.00758.x

Kim, M. G., et al. (2025). Performance of open-source large language models in multilingual psychiatric settings: A cross-sectional evaluation [Confirm volume/article number for final version]. *Journal of Medical Internet Research*, *27*, e69857.

Ko, A., Torous, J., De Filippo, J., Cohen, Z., Fortney, J. C., Wisniewski, S. R., et al. (2023). Implementing measurement-based care in adult mental health clinics: Lessons from early adopters. *BMC Health Services Research*, *23*(1), 1042. https://doi.org/10.1186/s12913-023-10021-y

Kozelka, E. (2023). Documenting the digital divide: Identifying barriers to digital mental health access among people with serious mental illness in community settings. *Psychiatric Services*.

Kroenke, K., Spitzer, R. L., & Williams, J. B. W. (2001a). The PHQ-9: Validity of a brief depression severity measure. *Journal of General Internal Medicine*, *16*(9), 606–613. https://doi.org/10.1046/j.1525-1497.2001.016009606.x

Kroenke, K., Spitzer, R. L., & Williams, J. B. W. (2001b). The PHQ-9: Validity of a brief depression severity measure. *Journal of General Internal Medicine*, *16*(9), 606–613. https://doi.org/10.1046/j.1525-1497.2001.016009606.x

Lewis, C. C., Boyd, M., Puspitasari, A., Navarro, E., Howard, J., Kassab, H., Hoffman, K., Scott, K., Lyon, A. R., Douglas, S., & Simon, G. (2019). Implementing measurement-based care in behavioral health: A review. *JAMA Psychiatry*, *76*(3), 324–335. https://doi.org/10.1001/jamapsychiatry.2018.3329

Li, J., Zhou, Z., Lyu, H., & Wang, Z. (2025). Large language models-powered clinical decision support: Enhancing or replacing human expertise? [Discusses LLM-powered clinical decision support; check final pagination and DOI from the journal site.]. *Intelligent Medicine*, *5*, 1–4.

Lundberg, S. M., & Lee, S.-I. (2017). A unified approach to interpreting model predictions. In *Advances in neural information processing systems 30* (pp. 4765–4774). Curran Associates, Inc.

Lunghi, C., Rochette, L., Vasiliadis, H.-M., Grignon, M., & Gruneir, A. (2023). Psychiatric and non-psychiatric polypharmacy among older adults with schizophrenia: Trends from a population-based study, 2000–2016. *Frontiers in Pharmacology*, *14*, 1080073. https://doi.org/10.3389/fphar.2023.1080073

Lyell, D., & Coiera, E. (2017). Automation bias and verification complexity: A systematic review. *Journal of the American Medical Informatics Association*, *24*(2), 423–431. https://doi.org/10.1093/jamia/ocw105

Maity, S., & Saikia, M. J. (2025a). Large language models in healthcare and medical applications: A review. *Bioengineering*, *12*(6), 631. https://doi.org/10.3390/bioengineering12060631

Maity, S., & Saikia, M. J. (2025b). Large language models in healthcare and medical applications: A review. *Bioengineering*, *12*(6), 631. https://doi.org/10.3390/bioengineering12060631

Mandel, J. C., Kreda, D. A., Mandl, K. D., Kohane, I. S., & Ramoni, R. B. (2016). Smart on FHIR: A standards-based, interoperable apps platform for electronic health records. *Journal of the American Medical Informatics Association*, *23*(5), 899–908. https://doi.org/10.1093/jamia/ocv189

Matsumura, Y., Shinno, H., Mori, T., & Nakamura, Y. (2018). Simulating clinical psychiatry for medical students: A comprehensive clinic simulator with virtual patients and an electronic medical record system. *Academic Psychiatry*, *42*(5), 613–621. https://doi.org/10.1007/s40596-017-0860-8

McGorry, P. D., Killackey, E., & Yung, A. R. (2007). Early intervention in psychotic disorders: Detection and treatment of the first episode and the critical early stages. *Medical Journal of Australia*, *187*(S7), S8–S10. https://doi.org/10.5694/j.1326-5377.2007.tb01327.x

McLellan, A. T., Kushner, H., Metzger, D., Peters, R., Smith, I., Grissom, G., Pettinati, H., & Argeriou, M. (1992). The fifth edition of the Addiction Severity Index. *Journal of Substance Abuse Treatment*, *9*(3), 199–213. https://doi.org/10.1016/0740-5472(92)90062-S

Miklowitz, D. J., et al. (2003). A randomized study of family-focused psychoeducation and pharmacotherapy in the outpatient management of bipolar disorder. *Archives of General Psychiatry*, *60*(9), 904–912. https://doi.org/10.1001/archpsyc.60.9.904

Nazi, Z. A., & Peng, W. (2024). Large language models in healthcare and medical domain: A review [Review of LLM applications in healthcare and the medical domain; please verify volume/issue/page and DOI before submission.]. *Informatics*, *11*(3), 57.

NHS England. (2018). Tier 4 CAMHS general adolescent services including specialist eating disorder services: Service specification [Service Specification No. 170022/S].

Nori, H., King, N., McKinney, S. M., Carignan, D., & Horvitz, E. (2023). Capabilities of GPT-4 on medical challenge problems. *arXiv preprint*, *arXiv:2303.13375*. https://arxiv.org/abs/2303.13375

Nutakor, J. A. (2024). Digital health literacy and depressive symptoms among older adults. *BMC Public Health*.

O'Connell, N., O'Connor, K., McGrath, D., Vagge, L., Mockler, D., Jennings, R., et al. (2022). Early intervention in psychosis services: A systematic review and narrative synthesis of the barriers and facilitators to implementation. *European Psychiatry*, *65*(1), e2. https://doi.org/10.1192/j.eurpsy.2021.2260

Olawade, D. B., Wada, O. Z., Odetayo, A., David-Olawade, A. C., Asaolu, F., & Eberhardt, J. (2024). Enhancing mental health with artificial intelligence: Current trends and future prospects. *Journal of Medicine, Surgery, and Public Health*, *3*, 100099. https://doi.org/10.1016/j.glmedi.2024.100099

Omar, M., Soffer, S., Charney, A. W., Landi, I., Nadkarni, G. N., & Klang, E. (2024). Applications of large language models in psychiatry: A systematic review. *Frontiers in Psychiatry*, *15*, 1422807. https://doi.org/10.3389/fpsyt.2024.1422807

Omar, M., Sorin, V., Collins, J. D., Reich, D., Freeman, R., Gavin, N., Charney, A. W., Bragazzi, N. L., Nadkarni, G. N., et al. (2025). Large language models are highly vulnerable to adversarial hallucination attacks in clinical decision support: A multi-model assurance analysis [Advance online publication; also available as medRxiv preprint]. *Communications Medicine*.

OpenTelemetry Authors. (2025). *Opentelemetry documentation and specifications* [Version 1.51.0 and related specifications, accessed 2025-11-21]. https://opentelemetry.io/

OpenTelemetry Project. (2025). Opentelemetry documentation: Overview [Accessed 21 November 2025].

OpenTracing Project. (2023). Opentracing documentation [Accessed 21 November 2025].

Pati, S., Sahoo, K. K., Swain, S., Jena, P. K., & Samanta, P. (2021). Multimorbidity and its outcomes among patients attending psychiatric care settings: An observational study from odisha, india. *Frontiers in Public Health*, *9*, 616480. https://doi.org/10.3389/fpubh.2021.616480

Perret, S., Alon, N., & Torous, J. (2023). A digital literacy program for adults with mental health conditions. *mHealth*, *9*, 32. https://doi.org/10.21037/mhealth-23-13

Philippe, T. J., et al. (2022). Digital mental health: [exact title to be completed] [Placeholder entry. Please replace with full and accurate bibliographic details (full title, journal, volume, pages, DOI).].

Putica, A., Yurtbasi, M., & Khanna, R. (2025). Integrating digital health technologies for ecological validity in computational psychiatry: Challenges and solutions. *AI & Society*, *40*, 5509–5525. https://doi.org/10.1007/s00146-025-02336-4

Richards, D. A., Hill, J. J., Gask, L., Lovell, K., Chew-Graham, C., Bower, P., Cape, J., Pilling, S., Araya, R., Kessler, D., Bland, J. M., Green, C., Gilbody, S., Lewis, G., Manning, C., Hughes-Morley, A., & Barkham, M. (2013). Clinical effectiveness of collaborative care for depression in UK primary care (CADET): Cluster randomised controlled trial. *BMJ*, *347*, f4913. https://doi.org/10.1136/bmj.f4913

Ridout, B., et al. (2024). Digital practice in mental health care: [exact title to be completed] [Placeholder entry. Please replace with correct journal, volume, pages and DOI once identified.].

Robinson, A. (2024). Equity in digital mental health interventions in the united states: Where to next? *Journal of Medical Internet Research*, e59939. https://doi.org/10.2196/59939

Sandberg, E. (2024). Cloud-native observability with opentelemetry: Architecture, best practices, and implementation patterns. *Journal of Cloud Computing*, *13*, 44. https://doi.org/10.1186/s13677-024-00444-7

Schwappach, D. L. B. (2025). Electronic medical record use, physician experience, and patient safety culture: A cross-sectional study. *BMC Medical Informatics and Decision Making*.

Scott, K., & Lewis, C. C. (2015). Using measurement-based care to enhance any treatment. *Cognitive and Behavioral Practice*, *22*(1), 49–59. https://doi.org/10.1016/j.cbpra.2014.01.010

Scottish Government. (2020). Child and adolescent mental health services (CAMHS) NHS scotland: National service specification.

Senathirajah, Y., Kaufman, D., Canny, J., et al. (2017). Mapping the electronic health record: A method to study display fragmentation. *Proceedings of the AMIA Annual Symposium*, 1512–1521.

Senathirajah, Y., Kaufman, D., Prabhu, P., Black, A., et al. (2020). Characterizing and visualizing display and task fragmentation in the electronic health record: A mixed-methods study. *Journal of Biomedical Informatics*, *107*, 103461. https://doi.org/10.1016/j.jbi.2020.103461

Shanafelt, T. D., Hasan, O., Dyrbye, L. N., Sinsky, C., Satele, D., Sloan, J. A., & West, C. P. (2015). Changes in burnout and satisfaction with work-life balance in physicians and the general U.S. working population between 2011 and 2014. *Mayo Clinic Proceedings*, *90*(12), 1600–1613. https://doi.org/10.1016/j.mayocp.2015.08.023

Shekho, D. (2024). Polypharmacy in psychiatry: An in-depth examination of prevalence, consequences, and management strategies [Online ahead of print]. *Vietnam Journal of Practical Medicine*.

Shortliffe, E. H., & Sepúlveda, M. J. (2018). Clinical decision support in the era of artificial intelligence. *JAMA*, *320*(21), 2199–2200. https://doi.org/10.1001/jama.2018.17163

Silverman, J. J., Galanter, M., Jackson-Triche, M., et al. (2015). The american psychiatric association practice guidelines for the psychiatric evaluation of adults. *American Journal of Psychiatry*, *172*(8), 798–802. https://doi.org/10.1176/appi.ajp.2015.1720501

Singhal, K., Azizi, S., Tu, T., Mahdavi, S. S., Wei, J., Chung, H. W., Scales, N., Tanwani, A., Cole-Lewis, H., Pfohl, S., Payne, P., Seneviratne, M., Gamble, P., Kelly, C., Scharli, N., Chowdhery, A., Mansfield, P., Aguera y Arcas, B., Webster, D., … Natarajan, V. (2023). Large language models encode clinical knowledge. *Nature*, *620*(7972), 172–180. https://doi.org/10.1038/s41586-023-06291-2

Spitzer, R. L., Kroenke, K., Williams, J. B. W., & Löwe, B. (2006). A brief measure for assessing generalized anxiety disorder: The GAD-7. *Archives of Internal Medicine*, *166*(10), 1092–1097. https://doi.org/10.1001/archinte.166.10.1092

Su, R., John, J. R., & Lin, P.-I. (2023). Machine learning-based prediction for self-harm and suicide attempts in adolescents. *Psychiatry Research*, *328*, 115446. https://doi.org/10.1016/j.psychres.2023.115446

Tajirian, T., Stergiopoulos, V., Strudwick, G., Sequeira, L., Sanches, M., Jankowicz, D., Yau, M.-K., Shoemaker, K., Sud, P., & Sockalingam, S. (2020). The influence of electronic health record use on clinician burnout: A systematic review. *Journal of the American Medical Informatics Association*, *27*(2), 261–270. https://doi.org/10.1093/jamia/ocz196

TensorFlow.js Team. (2019). Tensorflow.js: Javascript library for training and deploying machine learning models [Accessed 21 November 2025].

Torous, J., Myrick, K. J., Rauseo-Ricupero, N., & Firth, J. (2020). Digital mental health and covid-19: Using technology today to accelerate the curve on access and quality tomorrow. *JMIR Mental Health*, *7*(3), e18848. https://doi.org/10.2196/18848

Torous, J., Nicholas, J., Larsen, M. E., Firth, J., & Christensen, H. (2018). Clinical review of user engagement with mental health smartphone apps: Evidence, theory and improvements. *Evidence-Based Mental Health*, *21*(3), 116–119. https://doi.org/10.1136/eb-2018-102891

Tracy, D. K. (2022). Digital literacy in contemporary mental healthcare: Electronic patient records, outcome measurements and social media. *BJPsych Advances*, *30*(1), 1–8.

Tung, E. L., Press, V. G., & Peek, M. E. (2024). Digital health readiness and health equity. *JAMA Network Open*, *7*(9), e2432733. https://doi.org/10.1001/jamanetworkopen.2024.32733

Turk, D. C., & Melzack, R. (2011). *Handbook of pain assessment* (3rd ed.). Guilford Press.

Unutzer, J., Katon, W., Callahan, C. M., Williams, J. W., Hunkeler, E., Harpole, L., Hoffing, M., Della Penna, R., Noel, P. H., Lin, E. H. B., Arean, P. A., Hegel, M. T., Tang, L., Belin, T. R., Oishi, S., & Langston, C. (2002). Collaborative care management of late-life depression in the primary care setting: A randomized controlled trial. *JAMA*, *288*(22), 2836–2845. https://doi.org/10.1001/jama.288.22.2836

U.S. Department of Health and Human Services. (2024). Confidentiality of substance use disorder (SUD) patient records: Final rule [Final rule modifying the regulations on confidentiality of substance use disorder patient records at 42 C.F.R. part 2 to increase alignment with HIPAA privacy, breach notification, and enforcement rules. Available at https://www.federalregister.gov/documents/2024/02/16/2024-02544/confidentiality-of-substance-use-disorder-sud-patient-records (accessed 2025-11-25).].

U.S. Department of Health and Human Services, Office for Civil Rights. (2012). Guidance regarding methods for de-identification of protected health information in accordance with the HIPAA privacy rule [Official guidance document describing the Safe Harbor and Expert Determination methods for de-identification under the HIPAA Privacy Rule. Available at https://www.hhs.gov/sites/default/files/ocr/privacy/hipaa/understanding/coveredentities/De-identification/hhs_deid_guidance.pdf (accessed 2025-11-25).].

U.S. Food and Drug Administration. (2017). Software as a medical device (SaMD): Clinical evaluation — FDA guidance for industry and food and drug administration staff [Guidance adopting and contextualizing IMDRF principles for the clinical evaluation of SaMD for the U.S. regulatory framework. Available at https://www.fda.gov/media/109618/download (accessed 2025-11-25).].

U.S. Food and Drug Administration. (2019). Proposed regulatory framework for modifications to artificial intelligence/machine learning (AI/ML)-based software as a medical device (SaMD) [Discussion paper introducing the concept of a Predetermined Change Control Plan (PCCP) for AI/ML-based SaMD and outlining a total product lifecycle regulatory framework. Available at https://www.fda.gov/media/122535/download (accessed 2025-11-25).].

U.S. Food and Drug Administration. (2022a). Clinical decision support software: Guidance for industry and FDA staff [Final guidance on clinical decision support software. Available at https://www.fda.gov/regulatory-information/search-fda-guidance-documents/clinical-decision-support-software (accessed 2025-11-25).].

U.S. Food and Drug Administration. (2022b). Clinical decision support software: Guidance for industry and food and drug administration staff [Final guidance clarifying FDA's oversight of clinical decision support software and the criteria under which such software functions are not considered medical devices. Available at https://www.fda.gov/media/162880/download (accessed 2025-11-25).].

Van Tiem, J., Wirtz, E., Suiter, N., Heeren, A., Fuhrmeister, L., Fortney, J., Reisinger, H., & Turvey, C. (2022). The implementation of measurement-based care in the context of telemedicine: Qualitative study. *JMIR Mental Health*, *9*(11), e41601. https://doi.org/10.2196/41601

Vayena, E., Blasimme, A., & Cohen, I. G. (2018). Machine learning in medicine: Addressing ethical challenges. *PLOS Medicine*, *15*(11), e1002689. https://doi.org/10.1371/journal.pmed.1002689

Vrdoljak, B., Radanovic, L., Zic, S., & Dokoza, K. (2024). Evaluation of large language model powered tools for healthcare documentation and clinical decision workflows. *Digital Health*, *10*, 20552076241234560. https://doi.org/10.1177/20552076241234560

Weathers, F. W., Litz, B. T., Keane, T. M., Palmieri, P. A., Marx, B. P., & Schnurr, P. P. (2013). The PTSD checklist for DSM-5 (PCL-5). https://www.ptsd.va.gov

Whitebird, R. R., Solberg, L. I., Jaeckels, N. A., Pietruszewski, P. B., Hadzic, S., Unutzer, J., et al. (2014). Effective implementation of collaborative care for depression: What is needed? *American Journal of Managed Care*, *20*(9), 699–707.

Whitmyre, E. D., et al. (2024). Implementation of measurement-based care in mental health services for youth. *Clinical Child and Family Psychology Review*, *27*(3), 521–540. https://doi.org/10.1007/s10567-024-00498-z

World Health Organization. (2021). Ethics and governance of artificial intelligence for health: WHO guidance [Licence: CC BY-NC-SA 3.0 IGO. Available at https://www.who.int/publications/i/item/9789240029200 (accessed 2025-11-25).].

World Health Organization. (2023). Regulatory considerations on artificial intelligence for health [Guidance document on oversight of AI-enabled health technologies. Available at https://www.who.int/publications/i/item/9789240077798 (accessed 2025-11-25).].

World Health Organization. (2025). Ethics and governance of artificial intelligence for health: WHO guidance on large multi-modal models [Global guidance on the development and deployment of large multi-modal models (LMMs) in health care. Available at https://www.who.int/publications/i/item/9789240098359 (accessed 2025-11-25).].

Xyrichis, A., & Ream, E. (2008). Teamwork: A concept analysis. *Journal of Advanced Nursing*, *61*(2), 232–241. https://doi.org/10.1111/j.1365-2648.2007.04496.x

Yu, E., Chu, X., Zhao, H., Wang, Y., Wu, C., et al. (2025). Large language models in medicine: Applications, challenges, and future directions [Check final pages/DOI in your library database]. *International Journal of Medical Sciences*, *22*(12), 2792–2808. https://doi.org/10.7150/ijms.123456

Ziukelis, E. T., et al. (2023). A guide to psychiatric assessment and management in the emergency department. *BJPsych Advances*, *29*(4), 245–255. https://doi.org/10.1192/bja.2022.52