Capacity-Aware Software Design: A Pattern Language for Energy-Adaptive Applications

Dominikos Pritis
Independent Researcher
Kumbatio / entro314.labs, Athens, Greece
dominikos@kumbat.io

Preprint — DOI: 10.5281/zenodo.21915165

Abstract

We present a pattern language for designing software that adapts to the user's cognitive capacity. The patterns address a gap in current adaptive interface research: while systems commonly adapt to user skill level, device context, or accessibility needs, few treat cognitive energy—the fluctuating ability to handle complex, focused work—as a first-class design variable. We describe twelve patterns organized into four categories: state modeling, adaptation behavior, safety constraints, and system architecture. Each pattern is derived from the design and implementation of energy-system, a framework-agnostic library for energy-aware applications, and is presented with its context, forces, solution, consequences, and known failure modes. The patterns are intended to be reusable by any team building tools for variable-capacity users, independent of the specific library or framework.

Keywords: design patterns, adaptive interfaces, cognitive load, energy state, neurodivergent design, pattern language, capacity-aware systems

1 Introduction

Software adapts to many things: screen size, user preferences, accessibility requirements, network conditions, device capabilities. It rarely adapts to the user's cognitive state.

This is a significant gap. Cognitive capacity—the ability to handle complex decisions, sustain focused attention, and manage competing demands—varies substantially within a single person across a single day (Schmidt et al., 2007; Pritis, 2026b). For neurodivergent users, the variation is often larger and less predictable (Barkley, 1997). Yet most software presents the same interface, the same complexity, and the same demands regardless of whether the user is at peak cognitive capacity or approaching exhaustion.

Capacity-aware design is the practice of building software that recognizes and responds to this variation. It is related to, but distinct from, several established fields:

- Adaptive user interfaces adapt to user characteristics, but typically model stable traits (expertise level, preferences) rather than transient states (Brusilovsky, 2001).

- Accessibility design reduces barriers for users with disabilities, but typically through static accommodations rather than dynamic state-responsive behavior (Friedman and Bryen, 2007).
- Affective computing attempts to detect and respond to emotional states, but relies on inference rather than self-report (Picard, 1997).

This paper presents a pattern language—a structured set of reusable design solutions—for capacity-aware software. The patterns are derived from the design of energy-system, a TypeScript library for energy-aware applications whose underlying model is presented in Pritis (2026a), but are described at a level of abstraction that makes them applicable to any technology stack and application domain.

2 Pattern Language Overview

The twelve patterns are organized into four categories:

Table 1: Pattern categories and names.

Category	Patterns
State Modeling	Discrete Energy Levels
	Self-Declared State
	Timestamped Transitions
Adaptation Behavior	Strategy Decoupling
	Progressive Simplification
	Cognitive Profile Mapping
Safety Constraints	Escape Path Preservation
	Override Without Penalty
	Neutral State Language
	Staleness Detection
System Architecture	Framework-Agnostic Engine
	Pluggable Persistence

3 State Modeling Patterns

3.1 Pattern 1: Discrete Energy Levels

Context. A system needs to represent the user's current cognitive capacity so that other components can adapt behavior accordingly.

Forces. A continuous scale (0–100) provides maximum granularity but requires fine-grained judgment from the user. This judgment itself consumes cognitive resources, which are precisely what is scarce when the user needs the system most. A binary scale (on/off) is too coarse to drive meaningful differentiation.

Solution. Define a small set of discrete, named levels that map to recognizable cognitive states. Each level should correspond to a qualitatively different capacity for work. We use five levels: Peak (100), Active (75), Steady (50), Low (25), and Rest (0). The number is

small enough to require minimal decision effort and large enough to drive distinct adaptation behaviors at each level.

Consequences. The model trades precision for usability. Some users will find that their actual state falls between two levels. This is acceptable because the purpose is to drive appropriate adaptation, not to measure capacity with scientific accuracy.

Known failure mode. If level names carry value judgments (e.g., "Failing" instead of "Low"), the system penalizes honest self-assessment. See Pattern 9.

3.2 Pattern 2: Self-Declared State

Context. The system needs a source for the user's cognitive capacity level.

Forces. Biometric inference promises objectivity but delivers weak correlation with subjective cognitive state, particularly for neurodivergent users (Charles and Nixon, 2019; Barrett et al., 2019). Automated inference also transfers authority over self-description from the person to the system (Pritis, 2026c).

Solution. Make self-report the primary mechanism for setting energy state. The user explicitly declares their current level through a control in the interface. The system trusts this declaration.

Consequences. The system depends on user honesty and self-awareness. In practice, people are reasonably accurate about their own cognitive state. The cost of occasional inaccuracy is low compared to the cost of systematic misclassification from biometric inference.

3.3 Pattern 3: Timestamped Transitions

Context. The system stores energy state and needs to support pattern recognition, staleness detection, and historical analysis.

Forces. Storing only the current level discards temporal information. Storing raw event streams creates privacy and storage concerns.

Solution. Each state change records the level, the timestamp (epoch milliseconds), and the source (manual, scheduled, or inferred), together with two mechanical fields—a revision counter and a producer identity—that deterministically order concurrent writes across tabs, workers, or devices. This is the minimum data needed to support staleness detection, pattern analysis, provenance tracking, and cross-context reconciliation.

Consequences. Enables future features (energy pattern visualization, personalized schedule suggestions) without requiring additional data collection infrastructure.

4 Adaptation Behavior Patterns

4.1 Pattern 4: Strategy Decoupling

Context. The system needs to change application behavior based on energy level, but different applications need to respond differently.

Forces. Hardcoding behavior changes ("at level 25, hide the sidebar") couples the energy system to specific UI decisions. It also conflates the energy model with one particular response

to it.

Solution. Separate energy state from adaptation behavior. The energy system provides state. Consuming code defines strategies—pure functions that map energy levels to domain-specific configurations. A strategy carries a unique name and two methods: describe(level) returns a human-readable description; resolve(level) returns a typed configuration object.

Example strategies include UI Visibility (maps levels to chrome visibility), Notifications (maps levels to alert priority thresholds and batch intervals), Task Complexity (maps levels to maximum task complexity and break suggestions), Interaction Forgiveness (maps levels to undo windows, destructive-action confirmation, and autosave cadence), and Automation Autonomy (maps levels to how much latitude automated actions have before requiring user confirmation).

Consequences. An application can apply multiple strategies simultaneously. Each strategy is independently testable, composable, and replaceable.

4.2 Pattern 5: Progressive Simplification

Context. The UI needs to adapt as energy decreases, reducing cognitive load without losing functionality.

Forces. Abrupt changes are disorienting. A sudden layout shift when energy drops creates confusion at precisely the moment the user can least handle it. But if the interface remains identical at every level, the adaptation is meaningless.

Solution. Design adaptation as a gradient. Each step down in energy level removes one layer of visual or interactive complexity. At level 100, the full interface is available. At level 75, secondary chrome begins to reduce in prominence. At level 50, non-essential panels are hidden but accessible via toggle. At level 25, the interface reduces to primary content and essential navigation. At level 0, the interface enters a read-only, consumption-oriented mode.

Each transition should be smooth rather than instantaneous. The user should perceive the change as a coherent simplification, not a malfunction.

Consequences. The user experiences the interface as one continuous design that breathes, rather than as five discrete layouts that swap abruptly.

4.3 Pattern 6: Cognitive Profile Mapping

Context. Strategies need to make informed adaptation decisions, but different applications care about different aspects of cognitive capacity.

Forces. A single capacity number is insufficient for nuanced adaptation. The ability to sustain focus is different from decision-making capacity. But adding too many dimensions increases complexity.

Solution. Attach a cognitive profile to each energy level definition. The profile describes, for that level, the expected capacity along several dimensions: decision capacity, focus duration, task complexity tolerance, and interruption tolerance. Strategies read these profiles rather than the raw numeric level.

Consequences. The profile provides a bridge between the simple five-level model (which users interact with) and the nuanced adaptation decisions (which strategies implement).

5 Safety Constraint Patterns

5.1 Pattern 7: Escape Path Preservation

Context. The system simplifies the interface at lower energy levels. Some controls and navigation paths may be hidden.

Forces. Simplification reduces cognitive load. But removing access to critical functions at low energy creates a trap: the user is in a reduced state and has fewer tools to navigate out of it.

Solution. Define a set of escape paths that are never hidden, regardless of energy level: settings access, workspace/project switching, search or command palette, and the energy-level control itself. These paths may be visually de-emphasized but must remain discoverable and functional.

Consequences. The user always has a way to change their situation. This transforms the system from a constraining mode into a supportive one.

Known failure mode. Hiding the energy-level control itself at low energy. If the user cannot change their declared state, the system becomes a trap. The energy control must always be accessible.

5.2 Pattern 8: Override Without Penalty

Context. The system has adapted behavior based on declared energy state. The user disagrees with the adaptation.

Forces. If overriding is difficult, slow, or triggers a warning, the system is second-guessing the user. This is paternalistic and erodes trust.

Solution. Make overrides immediate, single-action, and consequence-free. No confirmation dialogs. No warnings. No logging of overrides as "anomalies."

Consequences. Override frequency becomes a useful signal for evaluating model fit (frequent overrides suggest the five-level model is not matching actual states), but this signal should be used for system improvement, not for evaluating the user.

5.3 Pattern 9: Neutral State Language

Context. The system needs labels, descriptions, and visual indicators for energy levels.

Forces. Language carries implicit value judgments. Labels like "Struggling" or "Failing" associate lower energy with personal deficiency. For users with depression—where shame is a clinical symptom—such framing amplifies cognitive burden.

Solution. Use descriptive, non-judgmental language. "Low" is a description. "Failing" is a judgment. "Rest" describes a state of consumption and recovery. "Burned out" implies damage. Each label should describe what the person can do at that level, not what they cannot.

Consequences. Users are more likely to report honestly when the system does not punish honesty with negative framing.

5.4 Pattern 10: Staleness Detection

Context. The user declared their energy level some time ago. Their actual capacity may have changed.

Forces. Frequent prompts to re-evaluate are themselves cognitive load. But acting on a stale declaration produces incorrect adaptation.

Solution. Use the timestamp from Pattern 3 to detect when a declared state is old (configurable threshold, e.g., 2–4 hours). Present a lightweight, dismissible prompt: "You set your energy level N hours ago. Still accurate?" The prompt does not change the level automatically.

Consequences. Balances the need for current state information against the cost of interruption. In energy-system, the engine exposes state-age metrics from which applications implement this pattern; the prompt itself is application-level behavior.

6 System Architecture Patterns

6.1 Pattern 11: Framework-Agnostic Engine

Context. The energy system needs to work across different application types: web, desktop, mobile, CLI, server-side.

Forces. Binding the energy model to a specific UI framework limits adoption and creates migration risk. But applications need framework-specific integration for practical use.

Solution. Implement the core engine (state management, strategy resolution, subscription, persistence) with no framework dependencies. Provide framework-specific bindings as separate, optional layers. The React layer wraps the engine in a context provider and hooks. A Vue layer would wrap it in composables. A CLI application uses the engine directly.

Consequences. The energy model is portable across any JavaScript/TypeScript environment. Teams can adopt the model incrementally.

6.2 Pattern 12: Pluggable Persistence

Context. Energy state needs to persist across sessions and potentially across devices.

Forces. Different runtime environments have different storage mechanisms. Hardcoding a single persistence mechanism limits deployment flexibility.

Solution. Define persistence as an interface with two methods: load() and save(state). Provide built-in implementations for common targets. Allow consumers to implement the interface for their own storage backend.

Consequences. The engine is decoupled from storage infrastructure. A web app uses localStorage. A desktop app uses SQLite. A team-sync application persists to a server. All use the same engine and strategies.

7 Pattern Relationships

The patterns are not independent. Key relationships:

- Discrete Energy Levels and Cognitive Profile Mapping together define the semantic model that strategies consume.
- Strategy Decoupling enables Progressive Simplification to be one strategy among many, rather than the system's sole behavior.
- Escape Path Preservation constrains Progressive Simplification: simplification must never hide escape paths.
- Override Without Penalty depends on Neutral State Language: if overriding triggers guilt-laden messaging, the override is not truly penalty-free.
- Staleness Detection depends on Timestamped Transitions for data.
- Self-Declared State is the foundation for Override Without Penalty: if state is inferred rather than declared, "override" becomes adversarial rather than corrective.

8 Evaluation

These patterns have not yet been empirically validated through controlled studies. The evaluation agenda includes:

1. Override frequency: How often do users change their declared level immediately after setting it? High frequency suggests the level model does not match lived experience.
2. Session continuation: Do users continue working longer (by choice, not pressure) when capacity-aware adaptation is active?
3. Time to first action: Does appropriate adaptation reduce the latency between session start and first meaningful work?
4. Focus duration: Does adaptation extend the duration of focused work sessions at each energy level?

9 Related Work

The pattern language format follows Alexander's original formulation (Alexander et al., 1977) and its adaptation to software by Gamma et al. (Gamma et al., 1994). Tidwell's Designing Interfaces (Tidwell, 2010) and van Welie and van der Veer's interaction design patterns (van Welie and van der Veer, 2003) provide precedent for UI-focused pattern languages in HCI.

Adaptive interface research has produced extensive work on adapting to user expertise (Brusilovsky, 2001) and device context, but we are not aware of prior pattern languages specifically addressing cognitive capacity as a transient, self-declared variable.

The neurodivergent design community has articulated principles for ADHD-friendly interfaces (Spiel et al., 2022), which inform several of our patterns. Our contribution is to formalize these concerns into a reusable pattern language with explicit forces, solutions, and failure modes.

10 Conclusion

Capacity-aware software design is not a niche concern. Cognitive energy varies for everyone. The twelve patterns described here provide a reusable vocabulary for teams building software that respects this variation. The patterns are technology-independent, domain-independent, and freely adoptable. They represent not a final answer but a structured starting point for a design practice that treats human cognitive variability as a first-class concern.

References

Alexander, C., Ishikawa, S., and Silverstein, M. (1977). A Pattern Language: Towns, Buildings, Construction. Oxford University Press.

Barkley, R. A. (1997). Behavioral inhibition, sustained attention, and executive functions: Constructing a unifying theory of ADHD. Psychological Bulletin, 121(1):65–94.

Barrett, L. F., et al. (2019). Emotional expressions reconsidered. Psychological Science in the Public Interest, 20(1):1–68.

Brusilovsky, P. (2001). Adaptive hypermedia. User Modeling and User-Adapted Interaction, 11(1):87–110.

Charles, R. L. and Nixon, J. (2019). Measuring mental workload using physiological measures. Applied Ergonomics, 74:221–232.

Friedman, M. G. and Bryen, D. N. (2007). Web accessibility design recommendations for people with cognitive disabilities. Technology and Disability, 19(4):205–212.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Picard, R. W. (1997). Affective Computing. MIT Press.

Pritis, D. (2026a). Energy as state, not time: Rethinking productivity models for variable-capacity work. Kumbatio / entro314.labs. https://doi.org/10.5281/zenodo.21915159.

Pritis, D. (2026b). The myth of the flat workday: What cognitive science already knows about variable capacity. Kumbatio / entro314.labs. https://doi.org/10.5281/zenodo.21915161.

Pritis, D. (2026c). Self-report over surveillance: The case for user-declared cognitive state. Kumbatio / entro314.labs. https://doi.org/10.5281/zenodo.21915163.

Schmidt, C., Collette, F., Cajochen, C., and Peigneux, P. (2007). A time to think: Circadian rhythms in human cognition. Cognitive Neuropsychology, 24(7):755–789.

Spiel, K., Hornecker, E., Williams, R. M., and Good, J. (2022). ADHD and technology research—investigated by neurodivergent readers. In Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '22). ACM.

Tidwell, J. (2010). Designing Interfaces. O'Reilly Media.

van Welie, M. and van der Veer, G. C. (2003). Pattern languages in interaction design. In Proc. Interact 2003, pages 527–534.