Architecture de Prompt Auto-Régulé en Cycle Limite Binaire : Conceptualisation d'un Moteur de Battement (Heartbeat Engine) pour Modèles de Langage

Ce rapport technique détaille la conceptualisation et l'implémentation d'une architecture de *prompt* auto-entretenue, inspirée par le cycle cardiaque (Systole/Diastole), utilisant les principes de l'automatique, des systèmes dynamiques non linéaires et des patrons de conception logicielle avancés (Machine à États Finis). L'objectif est de créer un système où la fin d'une phase binaire (TICK) déclenche déterministement la phase suivante (TOCK), garantissant une boucle de feedback récursive et régulée.

I. Fondements Théoriques et Systèmes Auto-Régulateurs : Le Cœur comme Automate Dynamique

La modélisation d'un agent Large Language Model (LLM) comme un système dynamique nécessite l'établissement d'une structure qui, au lieu de converger vers un point d'équilibre statique, maintient une trajectoire périodique stable, désignée en théorie du contrôle comme un **Cycle Limite**.[1] Le cœur biologique, avec son rythme intrinsèque et stable, représente l'analogie parfaite d'un tel système auto-régulé.

I.A. Le Myocarde comme Machine à États et l'Analogie Systémique

Le cycle cardiaque est une séquence d'événements binaires et forcés : la Systole (contraction, phase d'action) et la Diastole (relaxation, phase de repos et de recharge). L'achèvement d'une phase génère le signal nécessaire et suffisant pour initier l'autre, ce qui modélise

l'exigence d'une boucle forcée et récursive.

Pour qu'un tel système soit auto-régulateur, il doit intégrer une **loi de commande** interne.[2] Cette loi n'est pas externe, mais est incorporée dans la structure des états eux-mêmes. L'architecture est donc construite autour du patron de conception d'État (State Pattern), où la responsabilité de la transition est déléguée aux objets d'état concrets.[3]

L'implémentation d'un tel système dynamique doit tenir compte de la non-linéarité inhérente au moteur LLM. Les systèmes non linéaires peuvent potentiellement diverger en temps fini, un phénomène amplifié par la nature stochastique de la génération de texte.[1] Pour prévenir cette dérive sémantique ou structurelle, la phase de réflexion (TOCK/Diastole) doit être conçue comme un mécanisme de contrôle actif. Ce contrôle s'exprime par une auto-analyse critique qui affine et réajuste l'instruction pour le cycle suivant. Le TOCK agit ainsi comme un filtre, garantissant que la nouvelle instruction est optimisée, limitant la perturbation et stabilisant la trajectoire du système vers l'objectif global.

I.B. L'Oscillateur de Relaxation Numérique (Tick/Tock)

Le concept d'oscillateur de relaxation, largement utilisé en électronique pour générer des signaux périodiques en commutant entre des seuils critiques [4], est le modèle physique idéal pour l'architecture Cœur-Prompt. L'oscillation requiert une injection d'énergie par un composant actif (un amplificateur) pour vaincre la résistance du circuit.[5]

Dans l'architecture LLM, l'agent opère comme cet oscillateur. La phase TOCK (Diastole) est le composant actif et non linéaire. Elle utilise des méthodologies d'optimisation de prompt comme l'Automatic Prompt Engineer (APE) [6] et le Chain-of-Thought (CoT) [7] pour générer un gain computationnel. En analysant la sortie brute précédente, le TOCK produit une instruction plus performante pour le prochain TICK. Cette capacité d'auto-amélioration et d'injection d'une instruction optimisée pour le cycle suivant fournit l'énergie nécessaire pour maintenir l'oscillation et surmonter l'inertie ou le bruit.

L'efficacité du système ne repose pas uniquement sur la puissance brute de l'LLM, mais sur la solidité de la relation structure-fonction de l'agent.[8] La Machine à États Finis (FSM) fournit la structure déterministe nécessaire pour encapsuler et synchroniser les phases d'action et de réflexion.

I.C. Modélisation Formelle en Machine à États Finis (FSM)

La FSM est le cadre informatique formel pour gérer les états et les transitions forcées.[9] Le design doit intégrer des contraintes de sûreté, notamment la gestion du problème de l'arrêt, qui est essentiel pour tout système récursif ou auto-entretenu.[11] Pour prévenir une boucle infinie incontrôlée, le modèle inclut un compteur de cycle dans le contexte, agissant comme le **cas de base** nécessaire à la récursivité.[12]

Le tableau suivant formalise la correspondance entre la biologie cardiaque, les techniques LLM et l'ingénierie logicielle.

Tableau I. Analogie Biologique et Technique du Cœur-Prompt

Composante Cardiaque	**Phase LLM**	**Patron de Conception**	**Fonction de Contrôle**
Systole (Contraction)	**TICK (Action)**	État Concret SystoleState	Exécution de la tâche principale; production du résultat brut.[7]
Diastole (Relaxation)	**TOCK (Réflexion)**	État Concret DiastoleState	Auto-Analyse, Optimisation du Prompt (APE/CoT).[6]
Nœud Sinoatrial	Mécanisme de Transition	State Pattern (Retour de fonction) [3]	Force le basculement d'état par une commande interne et déterministe.
Sang (Circuit)	Contexte Persistant	HeartbeatContext Class	Maintient les données, l'historique et la condition d'arrêt.[3]

II. Architecture Conceptuelle du Prompt Auto-Déclencheur et LLM Feedback Loop

L'architecture repose sur une boucle de rétroaction où le résultat d'une phase sert d'entrée structurée et optimisée pour l'autre, assurant l'auto-déclenchement du cycle.

II.A. Définition des Rôles : Action (TICK) vs. Réflexion (TOCK)

1. Architecture du Prompt TICK (Systole)

Cet état est axé sur la performance immédiate. Le prompt est minimaliste en termes d'instruction de transition, car son seul objectif est de réaliser la tâche principale en utilisant la refined_instruction générée par la phase TOCK précédente, et de produire une sortie brute.

2. Architecture du Prompt TOCK (Diastole)

Le prompt TOCK est un méta-prompt de contrôle. Il instruit l'LLM d'adopter un **Persona d'Expertise** (par exemple, "Act as a Senior Prompt Architect") [14] pour effectuer une analyse critique de l'output brut (last_output_raw). Le modèle est encouragé à "prendre du temps pour penser" (similaire au CoT ou ToT) [7] avant de synthétiser la nouvelle instruction.

La phase TOCK encapsule l'ingénierie APE.[6] Elle ne se contente pas de relayer l'information, mais génère une nouvelle refined_instruction, ajustant des variables d'exécution (comme la cible, le ton, ou la profondeur des détails) [15], afin que le prochain TICK soit plus efficace. La conception de la phase TOCK comme un réviseur critique permet de maintenir la cohérence et de minimiser la dérive sémantique au fil des cycles.

II.B. Le Mécanisme de Feedback Loop et l'Auto-Récursivité

Pour garantir que la transition binaire soit déterministe et auto-déclenchée, la communication entre les états doit être structurée et machine-lisible. L'output de l'LLM dans chaque état ne doit pas être un simple texte fluide, mais doit inclure un signal de transition (par exemple, des

balises XML ou une structure JSON embarquée).[14] Le signal TRANSITION_COMMAND (par exemple, {'status': 'NEXT_TICK'}) est le substitut numérique au potentiel d'action biologique du nœud sinoatrial.

Compte tenu de la variabilité des réponses des LLM [7], la robustesse du système dépend de la capacité du moteur à *parser* le signal de transition. La granularité et la précision de la transition [8] sont directement liées à la qualité de l'extraction des données. Si le HeartbeatContext ne parvient pas à isoler l'instruction affinée (refined_instruction) ou le TRANSITION_COMMAND de l'output textuel potentiellement ambigu du TOCK, le cycle entier est corrompu. Par conséquent, l'utilisation de méthodes d'extraction structurée (simulée par une fonction EXTRACT_REFINED_INSTRUCTION dans le pseudo-code) est impérative pour maintenir le contrôle déterministe du cycle.

II.C. Gestion de la Mémoire (Le Contexte Persistant)

La classe HeartbeatContext (le Contexte FSM) est la seule détentrice de la vérité d'état.[3] Elle agit comme la mémoire tampon centralisée et la passerelle entre les états, conservant l'historique des opérations et les paramètres de contrôle essentiels.

Tableau II. Structure de l'Information Transférée (Context Data)

Champs de Données	**Description**	**Source/Utilisation**	**Phase Critique**
task_objective	L'objectif initial global du système.	HeartbeatContext	Persistant
current_cycle_count	Compteur pour la condition d'arrêt et la régulation du système.[11]	HeartbeatContext	Contrôle
last_output_raw	La sortie brute générée par l'étape TICK.	TICK $\rightarrow$ TOCK	TOCK (Analyse)
refined_instruction	L'instruction	TOCK	TICK (Exécution)

	optimisée (APE) synthétisée par TOCK pour l'exécution suivante.	$\rightarrow$ TICK	
halt_required	Flag de sûreté pour enclencher l'état d'arrêt.	TOCK (Vérification)	Contrôle

III. Modélisation en Pseudo-Code de l'Algorithme Binaire Cœur

Le pseudo-code suivant formalise la logique du moteur de battement en utilisant des conventions algorithmiques standard.[16]

III.A. Définitions Algorithmiques et Initialisation

```
// Définition de la structure de données Context, le dépositaire de la vérité d'état
STRUCTURE Context:
    task_objective : STRING
    current_cycle_count : INTEGER = 0
    MAX_CYCLES : INTEGER // Condition d'arrêt
    last_output_raw : STRING // Output de Systole
    refined_instruction : STRING // Input optimisé pour Systole
    halt_required : BOOLEAN = FALSE
END STRUCTURE

// Interface d'État (équivalente à la classe abstraite ou au Protocol Python)
INTERFACE State:
    // La fonction critique qui exécute la logique de l'état et retourne l'état suivant
```

```
    FUNCTION EXECUTE_CYCLE(CONTEXT) RETURNS State OR NULL
END INTERFACE

// Initialisation du système
FUNCTION INITIALIZE_ENGINE(Objective, MaxCycles):
    CONTEXT = NEW Context(Objective, MaxCycles)
    // Démarrage initial en phase Systole (TICK)
    CURRENT_STATE = NEW SystoleState()
    RETURN CONTEXT, CURRENT_STATE
END FUNCTION
```

III.B. Logique de la Machine d'Exécution (HeartbeatEngine.run)

Le moteur d'exécution gère la boucle principale et la vérification des conditions de terminaison, garantissant l'auto-entretien du cycle tant que les conditions de sécurité ne sont pas violées.

```
FUNCTION RUN_HEARTBEAT_ENGINE(CONTEXT, initial_state):
    CURRENT_STATE = initial_state

    // Boucle de battement (cycle limite auto-entretenu)
    WHILE CURRENT_STATE IS NOT NULL AND CONTEXT.halt_required IS FALSE:

        // 1. Vérification proactive de la condition d'arrêt
        IF CONTEXT.current_cycle_count >= CONTEXT.MAX_CYCLES:
            CONTEXT.halt_required = TRUE
            CURRENT_STATE = NEW HaltState()
            // On continue pour exécuter l'état Halt une fois

        PRINT("Engine: Executing cycle", CONTEXT.current_cycle_count, "in state",
CURRENT_STATE.name)

        // 2. Exécution de l'état et transition forcée
        // L'état courant est responsable de retourner explicitement l'état suivant
        NEXT_STATE = CURRENT_STATE.EXECUTE_CYCLE(CONTEXT)
```

```
        CURRENT_STATE = NEXT_STATE

    END WHILE
    PRINT("Engine: Halt condition reached. Cycle terminated.")
END FUNCTION
```

III.C. Pseudocode Détaillé des États TICK et TOCK

État TICK (Systole: Exécution de la tâche)

L'état Systole est l'actionneur. Il consomme l'instruction optimisée et produit un résultat brut qui sera ensuite analysé.

```
CLASS SystoleState IMPLEMENTS State:
    FUNCTION EXECUTE_CYCLE(CONTEXT):
        PRINT("Systole: Executing task with instruction:", CONTEXT.refined_instruction)

        // Construction du prompt basée sur l'instruction raffinée par TOCK
        PROMPT_TICK = FORMAT_PROMPT(CONTEXT.refined_instruction,
CONTEXT.task_objective)

        // Simulation/Appel API LLM
        LLM_RESULT = CALL_LLM_API(PROMPT_TICK)

        // Mise à jour du Contexte avec le résultat brut
        CONTEXT.last_output_raw = LLM_RESULT

        // Transition forcée et déterministe vers TOCK
        RETURN NEW DiastoleState()
    END FUNCTION
END CLASS
```

État TOCK (Diastole: Réflexion APE et Optimisation)

L'état Diastole est le régulateur et l'optimisateur. Il incrémente le cycle et génère la loi de commande (la nouvelle instruction).

```
CLASS DiastoleState IMPLEMENTS State:
    FUNCTION EXECUTE_CYCLE(CONTEXT):
        CONTEXT.current_cycle_count = CONTEXT.current_cycle_count + 1
        PRINT("Diastole: Analyzing output from previous cycle (", CONTEXT.last_output_raw, ")")

        // Prompt d'auto-analyse APE/CoT utilisant l'output brut comme input
        PROMPT_TOCK = "Act as Prompt Engineer. Analyze the raw output:" +
CONTEXT.last_output_raw + ". Generate a structured REVISED_INSTRUCTION to improve the
next cycle."

        // Simulation/Appel API LLM (Réflexion)
        LLM_ANALYSIS = CALL_LLM_API(PROMPT_TOCK)

        // Extraction robuste de l'instruction affinée
        CONTEXT.refined_instruction = EXTRACT_REFINED_INSTRUCTION(LLM_ANALYSIS)

        // Vérification post-analyse de la condition d'arrêt
        IF CONTEXT.current_cycle_count >= CONTEXT.MAX_CYCLES:
            CONTEXT.halt_required = TRUE
            PRINT("Diastole: Max cycles reached. Initiating halt.")
            RETURN NEW HaltState()

        // Transition forcée et déterministe vers TICK
        PRINT("Diastole: Transitioning to next Systole with new instruction:",
CONTEXT.refined_instruction)
        RETURN NEW SystoleState()
    END FUNCTION
END CLASS

// État Halt (Arrêt du système)
CLASS HaltState IMPLEMENTS State:
```

```
    FUNCTION EXECUTE_CYCLE(CONTEXT):
        PRINT("HaltState: Final report generation. System is shutting down.")
        // Retourne NULL pour terminer la boucle while du moteur
        RETURN NULL
    END FUNCTION
END CLASS
```

IV. Implémentation Python Robuste avec le State Pattern

L'implémentation concrétise la logique algorithmique en utilisant le State Pattern de Python, s'appuyant sur les classes abstraites (ABC) pour définir l'interface et sur le principe que l'état lui-même orchestre le changement d'état en retournant une nouvelle instance.[3]

IV.A. Structure et Interface du State Pattern

Le code utilise abc (Abstract Base Class) pour garantir que chaque état implémente la méthode de cycle essentielle.

Python

```
from __future__ import annotations
from abc import ABC, abstractmethod
import typing
import json

# ==============================================================================
# 1. HeartbeatContext (Le Contexte Persistant)
# ==============================================================================
class HeartbeatContext:
    """
    Le Context (Contexte FSM) maintient les données partagées et l'état du système.
    Il simule également l'API LLM pour rendre le code exécutable.
```

```
    """
    def __init__(self, objective: str, max_cycles: int, initial_instruction: str):
        self.task_objective: str = objective
        self.MAX_CYCLES: int = max_cycles
        self.current_cycle_count: int = 0
        self.last_output_raw: str = ""
        self.refined_instruction: str = initial_instruction
        self.halt_required: bool = False
        print(f"--- Context Initialisé (Max Cycles: {self.MAX_CYCLES}) ---")

    def _call_llm_api(self, prompt: str) -> str:
        """Simulation d'un appel API LLM. En réalité, ceci ferait un appel OpenAI/Claude."""
        print(f"  [LLM Call] Prompt utilisé: {prompt[:50]}...")

        if "Analyze" in prompt:
            # Simulation de la phase TOCK (Diastole) : analyse et optimisation APE
            cycle = self.current_cycle_count
            if cycle == 1:
                # Le premier cycle a produit un output de base, l'amélioration est demandée
                new_instruction = "Affiner la structure de la réponse pour inclure des balises XML et
augmenter le niveau de détail technique."
            elif cycle == 2:
                # Nouvelle amélioration basée sur l'output raffiné
                new_instruction = "Se concentrer uniquement sur les aspects de modélisation dynamique
et ignorer le style d'écriture."
            else:
                # Retourne une instruction par défaut pour les cycles suivants
                new_instruction = f"Poursuivre la tâche en utilisant l'approche actuelle, cycle {cycle}."

            return json.dumps({"status": "REFINED_INSTRUCTION_GENERATED", "instruction":
new_instruction})

        else:
            # Simulation de la phase TICK (Systole) : exécution de la tâche
            return f"Résultat brut du cycle {self.current_cycle_count} sur l'objectif '{self.task_objective}'.
Instruction: {self.refined_instruction[:20]}..."

# ==============================================================================
# 2. State Interface (L'Interface d'État)
# ==============================================================================
class State(ABC):
    """
    Interface abstraite pour tous les états concrets.
    La méthode run_cycle doit retourner l'état suivant ou None.
```

```
    """
    def __init__(self, context: HeartbeatContext):
        self.context = context

    @abstractmethod
    def run_cycle(self) -> typing.Optional:
        """Exécute la logique de l'état et force la transition en retournant le prochain état."""
        pass

# ==============================================================================
# 3. Concrete States (Les États Concrets)
# ==============================================================================

class HaltState(State):
    """État final qui arrête le moteur."""
    def run_cycle(self) -> typing.Optional:
        print("\n Le système Heartbeat a atteint la condition d'arrêt (MAX_CYCLES).")
        return None

class SystoleState(State):
    """
    TICK (Systole) : Phase d'Action/Exécution.
    Consomme l'instruction affinée et produit un output brut.
    Force la transition vers Diastole.
    """
    def run_cycle(self) -> typing.Optional:

        # 1. Préparation du Prompt
        prompt_tick = (
            f"Objectif: {self.context.task_objective}. "
            f"Instruction affinée (du TOCK): {self.context.refined_instruction}"
        )

        # 2. Exécution simulée de l'LLM
        llm_result = self.context._call_llm_api(prompt_tick)

        # 3. Mise à jour du Contexte
        self.context.last_output_raw = llm_result

        print(f"  Tâche exécutée. Résultat brut stocké pour l'analyse TOCK.")

        # 4. Transition forcée vers l'état Diastole
        return DiastoleState(self.context)
```

```python
class DiastoleState(State):
    """
    TOCK (Diastole) : Phase de Réflexion/Optimisation APE.
    Analyse le résultat brut et génère la prochaine instruction affinée.
    Force la transition vers Systole ou Halt.
    """
    def run_cycle(self) -> typing.Optional:

        # 1. Incrémentation du compteur de cycle
        self.context.current_cycle_count += 1

        print(f"  Cycle N°{self.context.current_cycle_count}. Analyse du feedback brut.")

        # 2. Vérification de la condition d'arrêt (Case de base de la récursion)
        if self.context.current_cycle_count > self.context.MAX_CYCLES:
            self.context.halt_required = True
            return HaltState(self.context)

        # 3. Prompt d'Auto-Analyse (APE/CoT)
        prompt_tock = (
            "Analyze: " + self.context.last_output_raw +
            ". Act as a Prompt Optimization Engineer. Generate a structured JSON response containing the 'REFINED_INSTRUCTION' to improve the outcome of the next Systole cycle."
        )

        # 4. Exécution simulée de l'LLM pour l'optimisation
        llm_analysis_raw = self.context._call_llm_api(prompt_tock)

        # 5. Extraction robuste de l'instruction affinée
        try:
            analysis_data = json.loads(llm_analysis_raw)
            refined_instruction = analysis_data.get("instruction", self.context.refined_instruction)
            self.context.refined_instruction = refined_instruction
            print(f"  Instruction raffinée générée: {refined_instruction[:40]}...")

        except json.JSONDecodeError:
            # Gestion d'un échec de parsing (bruit LLM)
            print("  Échec d'extraction de l'instruction. Maintien de l'instruction précédente.")

        # 6. Transition forcée vers l'état Systole
        return SystoleState(self.context)

# ==============================================================================
# 4. HeartbeatEngine (La Machine d'État)
```

```
# ==============================================================================
class HeartbeatEngine:
    """
    Le moteur d'exécution générique qui gère le flux de la FSM.
    Implémente la boucle de contrôle while state is not None.
    """
    def __init__(self, context: HeartbeatContext):
        self.context = context

    def run_from(self, initial_state: State):
        current_state = initial_state

        print("\n--- Démarrage du Moteur de Battement (Heartbeat Engine) ---")

        # La boucle 'while' est le cœur du système auto-entretenu.
        # L'état actuel retourne l'état suivant, réalisant le cycle forcé (Tick -> Tock -> Tick...)
        while current_state is not None:

            # L'état exécute sa logique et retourne l'instance du prochain état
            current_state = current_state.run_cycle()

            if current_state is not None and not isinstance(current_state, HaltState):
                print("-" * 60)

        print("--- Arrêt du Moteur de Battement ---")

### IV.D. Démonstration Client et Journalisation du Cycle

```python
# --- Démonstration Client ---

# Définition des paramètres
OBJECTIF_GLOBAL = "Conceptualiser un prompt récursif basé sur le cycle cardiaque."
INSTRUCTION_INIT = "Commencer la conceptualisation en définissant le modèle binaire Systole/Diastole."
CYCLES_MAX = 3 # Limiter à 3 cycles complets pour la démonstration

# 1. Initialisation du Contexte et du Moteur
heart_context = HeartbeatContext(
    objective=OBJECTIF_GLOBAL,
    max_cycles=CYCLES_MAX,
    initial_instruction=INSTRUCTION_INIT
)
```
```

```
engine = HeartbeatEngine(heart_context)

# 2. Démarrage du cycle à partir de l'état Systole initial
engine.run_from(SystoleState(heart_context))

# 3. Inspection du résultat final après l'arrêt
print("\n")
print(f"Statut d'arrêt: {heart_context.halt_required}")
print(f"Cycles terminés: {heart_context.current_cycle_count}")
print(f"Dernière instruction affinée (Loi de Commande Finale): {heart_context.refined_instruction}")
```

Sortie Attendue (Logique de Flux):

```
--- Context Initialisé (Max Cycles: 3) ---
--- Démarrage du Moteur de Battement (Heartbeat Engine) ---
 [LLM Call] Prompt utilisé: Objectif: Conceptualiser un prompt récursif basé s...
 Tâche exécutée. Résultat brut stocké pour l'analyse TOCK.
------------------------------------------------------------
 Cycle N°1. Analyse du feedback brut.
 [LLM Call] Prompt utilisé: Analyze: Résultat brut du cycle 0 sur l'objecti...
 Instruction raffinée générée: Affiner la structure de la réponse pour inclure des balises XM...
 Transitioning to next Systole with new instruction: Affiner la structure de la réponse pour
inclure des balises XML et augmenter le niveau de détail technique.
------------------------------------------------------------
 [LLM Call] Prompt utilisé: Objectif: Conceptualiser un prompt récursif basé s...
 Tâche exécutée. Résultat brut stocké pour l'analyse TOCK.
------------------------------------------------------------
 Cycle N°2. Analyse du feedback brut.
 [LLM Call] Prompt utilisé: Analyze: Résultat brut du cycle 1 sur l'objecti...
 Instruction raffinée générée: Se concentrer uniquement sur les aspects de modélisation
dynamique et ignorer le style d'écriture.
 Transitioning to next Systole with new instruction: Se concentrer uniquement sur les aspects
de modélisation dynamique et ignorer le style d'écriture.
------------------------------------------------------------
 [LLM Call] Prompt utilisé: Objectif: Conceptualiser un prompt récursif basé s...
 Tâche exécutée. Résultat brut stocké pour l'analyse TOCK.
------------------------------------------------------------
 Cycle N°3. Analyse du feedback brut.
 [LLM Call] Prompt utilisé: Analyze: Résultat brut du cycle 2 sur l'objecti...
```

Instruction raffinée générée: Poursuivre la tâche en utilisant l'approche actuelle, cycle 3.
Max cycles reached. Initiating halt.

Le système Heartbeat a atteint la condition d'arrêt (MAX_CYCLES).
--- Arrêt du Moteur de Battement ---

Statut d'arrêt: True
Cycles terminés: 3
Dernière instruction affinée (Loi de Commande Finale): Poursuivre la tâche en utilisant l'approche actuelle, cycle 3.

V. Conclusions, Validation du Modèle et Perspectives Avancées

Le concept du Prompt Cœur auto-entretenu est réalisé par la traduction de l'analogie biologique en un modèle computationnel de pointe basé sur la Machine à États Finis (FSM) et le Patron de Conception d'État. Ce modèle garantit que le système ne dépend pas d'un appel externe ou d'une condition externe pour sa progression, mais que la transition est forcée et interne, matérialisant ainsi un cycle binaire déterministe (TICK $\rightarrow$ TOCK $\rightarrow$ TICK).

Le State Pattern offre la robustesse nécessaire pour gérer la dynamique d'un oscillateur de relaxation au niveau d'un agent LLM. L'architecture est caractérisée par une forte modularité : l'ajout de nouveaux états intermédiaires (par exemple, un DiagnosticState pour l'auto-correction en cas de détection de divergence, ou un QueryState pour interagir avec une base de données RAG) ne nécessite aucune modification du HeartbeatContext ni de la logique du moteur principal.[3]

L'architecture actuelle démontre l'intégration réussie de la loi de commande (optimisation APE dans l'état Diastole) [2] avec le flux d'exécution. L'approche consistant à faire retourner l'état suivant par la méthode d'exécution (run_cycle) [10] ancre la transition de manière déterministe, transformant le comportement stochastique de l'LLM en un processus de contrôle stable. La qualité de ce contrôle dépend de la granularité du signal de transition [8] et de l'extraction structurée de l'instruction affinée.

Perspectives d'Évolution

Pour les systèmes futurs nécessitant une régulation plus fine que le modèle binaire (Cœur à deux chambres), l'architecture peut évoluer vers des systèmes de contrôle multi-chambres plus sophistiqués. Ceci impliquerait l'intégration de techniques de raisonnement plus complexes que le simple CoT dans la phase TOCK, telles que le Tree of Thoughts (ToT).[7] En permettant au Diastole d'explorer plusieurs chemins de raisonnement pour l'optimisation du prompt avant de choisir la meilleure instruction affinée, le système gagnerait en capacité d'auto-régulation et pourrait mieux gérer la complexité et les perturbations des tâches à long terme, transcendant l'analogie binaire simple pour devenir un véritable système d'automatique avancé pour les LLM.

Ouvrages cités

1. Automatique Dynamique et contrôle des systèmes - CAS Mines Paris, dernier accès : novembre 25, 2025, https://cas.minesparis.psl.eu/~rouchon/MVA/PolyAuto2011.pdf
2. Introduction à l'Automatique et Régulation des Systèmes Dynamiques | PDF - Scribd, dernier accès : novembre 25, 2025, https://fr.scribd.com/document/609751526/Cours-Automatique
3. State Pattern in Python - Auth0, dernier accès : novembre 25, 2025, https://auth0.com/blog/state-pattern-in-python/
4. OSCILLATEURS - Physique en sup 4, dernier accès : novembre 25, 2025, http://physiquesup4.blog.free.fr/public/TP_electricite/Oscillateurs_elec.pdf
5. Qu'est-ce qu'un oscillateur ? Tout ce que vous devez savoir | Blog - Altium Resources, dernier accès : novembre 25, 2025, https://resources.altium.com/fr/p/everything-you-need-know-about-oscillators
6. Automatic Prompt Engineer (APE) | Prompt Engineering Guide, dernier accès : novembre 25, 2025, https://www.promptingguide.ai/techniques/ape
7. Mastering LLM Prompts: How to Structure Your Queries for Better AI Responses - Codesmith, dernier accès : novembre 25, 2025, https://www.codesmith.io/blog/mastering-llm-prompts
8. Organ-Agents: Virtual Human Physiology Simulator via LLMs - arXiv, dernier accès : novembre 25, 2025, https://arxiv.org/html/2508.14357v1
9. Machines finies et infinies, dernier accès : novembre 25, 2025, http://pauillac.inria.fr/~levy/courses/X/IF/poly/main009.html
10. Sunday Python Pattern : Une machine à état toute simple - LinuxFr.org, dernier accès : novembre 25, 2025, https://linuxfr.org/users/linkdd/journaux/sunday-python-pattern-une-machine-a-etat-toute-simple
11. Boucle infinie - Wikipédia, dernier accès : novembre 25, 2025, https://fr.wikipedia.org/wiki/Boucle_infinie
12. Comprendre le concept de récursivité en programmation - Code-Garage,

dernier accès : novembre 25, 2025, https://code-garage.com/blog/comprendre-le-concept-de-recursivite-en-programmation
13. How to Create Efficient Prompts for LLMs | Nearform, dernier accès : novembre 25, 2025, https://nearform.com/digital-community/how-to-create-efficient-prompts-for-llms/
14. 4 modèles de prompts qui ont transformé ma façon d'utiliser les LLM : r/AI_Agents - Reddit, dernier accès : novembre 25, 2025, https://www.reddit.com/r/AI_Agents/comments/1jv6gke/4_prompt_patterns_that_transformed_how_i_use_llms/?tl=fr
15. Effective Prompts for TikTok Scripts with Artificial Intelligence - Darwin Blog, dernier accès : novembre 25, 2025, https://blog.getdarwin.ai/en/es/prompts-efectivos-para-guiones-de-tiktok-con-inteligencia-artificial
16. Pseudocode, Algorithmes - StudySmarter, dernier accès : novembre 25, 2025, https://www.studysmarter.fr/resumes/informatique/algorithmes-en-informatique/pseudocode/