Soviet Chess: A Multi-Agent Framework for AI-Assisted Software Development with Reinforcement Learning

ES Soviet Chess: Un Framework Multi-Agente para Desarrollo de Software Asistido por IA con Aprendizaje Reforzado

David Pineda

Sumud Project — April 2026

Abstract

EN We present Soviet Chess, a three-layer multi-agent framework for software development where six specialized AI agents — modeled after chess pieces — collaborate through a blackboard architecture to build, test, and deploy software. The system addresses three failure modes observed in single-agent AI coding: context loss during task delegation, error accumulation from deferred compilation, and role confusion from monolithic prompting. An offline reinforcement learning module analyzes daily session logs and tunes agent strategy weights using contextual bandits, optimizing for token efficiency, compilation error rate, and code quality metrics. Validated on a 51K-line Rust/TypeScript codebase, the framework reduced per-file compilation errors from ~28 to <0.5 and improved code health scores by 20% over 5 sessions.

ES Presentamos Soviet Chess, un framework multi-agente de tres capas para desarrollo de software donde seis agentes IA especializados — modelados como piezas de ajedrez — colaboran mediante una arquitectura de pizarra (blackboard) para construir, testear y desplegar software. El sistema aborda tres modos de fallo observados en codificacion con un solo agente: perdida de contexto en delegacion de tareas, acumulacion de errores por compilacion diferida, y confusion de roles en prompting monolitico. Un modulo de aprendizaje reforzado offline analiza logs de sesion diarios y ajusta pesos de estrategia usando bandidos contextuales, optimizando eficiencia de tokens, tasa de errores de compilacion y metricas de calidad de codigo. Validado en un codebase de 51K lineas Rust/TypeScript, el framework redujo errores de compilacion por archivo de ~28 a <0.5 y mejoro scores de salud del codigo en 20% en 5 sesiones.

Keywords: multi-agent systems, LLM-assisted development, reinforcement learning, software engineering, code quality

1. Introduction / Introduccion

EN Large Language Models (LLMs) have transformed software development through tools like GitHub Copilot, Cursor, and Claude Code. However, a single AI agent handling architecture, implementation, testing, and deployment simultaneously produces characteristic failure patterns:

ES Los Modelos de Lenguaje (LLMs) han transformado el desarrollo de software con herramientas como GitHub Copilot, Cursor y Claude Code. Sin embargo, un solo agente IA manejando arquitectura, implementacion, testing y deployment produce patrones de fallo caracteristicos:

1. **Context Loss / Perdida de contexto**: When subtasks are delegated to sub-agents, each loses visibility of the whole system. In our observation, this caused 138 compilation errors from incorrect import guessing during a module split. / Cuando las subtareas se delegan a sub-agentes, cada uno pierde visibilidad del sistema completo.

2. **Error Accumulation / Acumulacion de errores**: Writing multiple files before compiling allows errors to compound. One wrong import in file 1 cascades to 138 errors across 5 files. / Escribir multiples archivos antes de compilar permite que los errores se multipliquen.
3. **Role Confusion / Confusion de roles**: A monolithic agent optimizes for the last instruction rather than holistic quality. Qian et al. (2023) showed that role separation reduces hallucination by 66%. / Un agente monolitico optimiza para la ultima instruccion en vez de la calidad global.

EN We address these with a structured multi-agent system inspired by Minsky's Society of Mind (1986), chess organizational metaphor, and blackboard architecture (Hayes-Roth, 1985).

2. Architecture / Arquitectura

2.1 Three-Layer Model / Modelo de tres capas

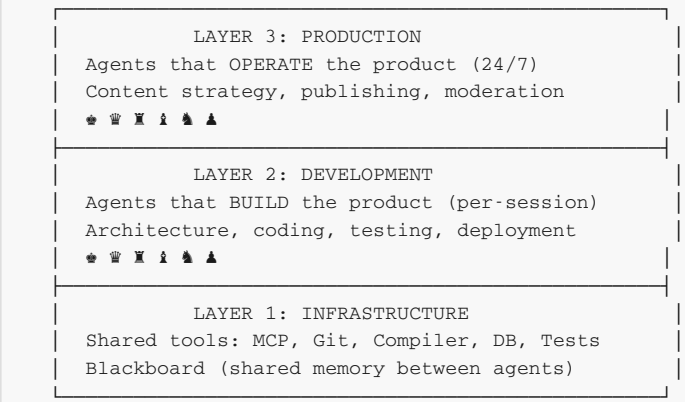

Figure 1: Three-layer architecture. Each layer uses the same 6 chess-piece roles for different domains. / Figura 1: Arquitectura de tres capas.

2.2 Agent Roles / Roles de agentes

Piece	Role	Layer 2 (Dev)	Layer 3 (Prod)	Can Write Code
♚ Rey	Architect/Strategist	Task decomposition, ADRs	Content strategy	No
♛ Reina	Reviewer/Gate	Code review, security	Ethical gate	No
♜ Torre	Backend/Publisher	Rust/Python/Go code	Channel publishing	Yes
♝ Alfil	Frontend/Engagement	Vue/React/TS code	Community mgmt	Yes
♞ Caballo	Tester/Analyst	Tests, coverage	Sentiment analysis	Tests only
♟ Peon	DevOps/Maintenance	Deploy, CI/CD	Data sync, cron	Scripts only

Table 1: Agent roles across layers. Minimal authority principle enforced. / Tabla 1: Roles de agentes por capa.

> **Design Principle / Principio de diseno:** Each agent has the minimum capabilities for its role (Saltzer & Schroeder, 1975). Torre cannot deploy. Caballo cannot write production code. This prevents entire classes of errors. / Cada agente tiene las capacidades minimas para su rol.

3. Blackboard Communication / Comunicacion por Pizarra

EN Agents communicate exclusively through a shared blackboard (Hayes-Roth, 1985), never directly. This ensures full traceability and prevents context loss.

ES Los agentes se comunican exclusivamente a traves de una pizarra compartida, nunca directamente. Esto asegura trazabilidad completa y previene perdida de contexto.

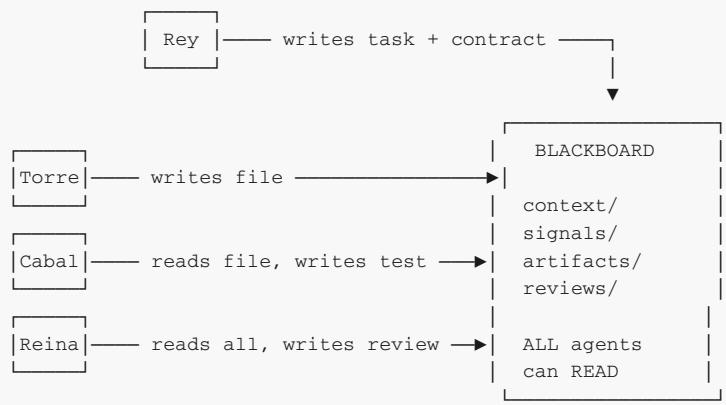

Figure 2: Blackboard communication pattern. / Figura 2: Patron de comunicacion por pizarra.

3.1 Import Map Protocol / Protocolo de mapa de imports

EN Before any module split, Torre writes a dependency map to the blackboard. This prevents the import-guessing problem that caused 138 errors in our initial approach.

```
BEFORE splitting a 3,675-line file:

Step 1: Torre reads entire file
Step 2: Torre extracts → import_map.json:
        {
          "external_imports": ["sumud_db::repo::travelers", ...],
          "public_functions": ["handle_message", ...],
          "cross_references": {"handle_command": ["handle_admin_command", ...]}
        }
Step 3: Torre writes import_map to blackboard
Step 4: For EACH submodule:
        a) Read import_map from blackboard
        b) Write ONE file
        c) cargo check  ← COMPILE IMMEDIATELY
        d) Fix any errors (typically 0-2)
        e) Next file
```

Figure 3: Import Map Protocol prevents cascading compilation errors. / Figura 3: El protocolo de mapa de imports previene errores en cascada.

3.2 Task Contracts / Contratos de tareas

EN Every task includes machine-verifiable pre/postconditions (Meyer, 1992). The critical postcondition is `cargo check = success` after every file write.

```
task: "Split channels.rs into submodules"
preconditions:
  - cargo check: success
```

```
    - file channels.rs: exists, 3438 lines
  postconditions:
    - cargo check: success          ← MANDATORY
    - cargo test: all pass          ← MANDATORY
    - channels.rs: deleted
    - channels/: exists, 5 files
    - max file: ≤1500 lines
    - all original functions: preserved
```

4. Generate-Check-Fix Loop / Ciclo Generar-Verificar-Corregir

EN The core innovation: treating the compiler as a fast feedback oracle (2-5 seconds per check) and invoking it after every single file operation.

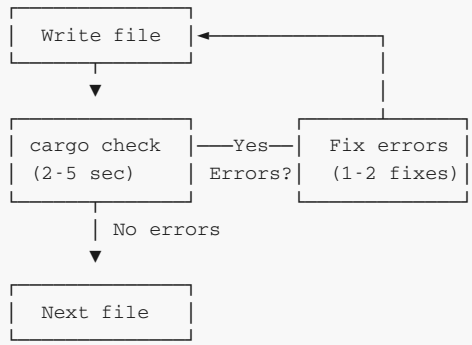

Figure 4: Generate-Check-Fix loop. Adapted from TDD's Red-Green-Refactor (Beck, 2003). / Figura 4: Ciclo Generar-Verificar-Corregir.

EN **Without the loop:**

- 5 files written blindly
- 138 compilation errors
- 30 minutes debugging imports
- Cross-file error propagation

ES **Con el ciclo:**

- 5 files escritos con verificacion
- 0-4 errores totales
- 25 segundos en verificaciones
- Errores aislados por archivo

5. Reinforcement Learning Module / Modulo de Aprendizaje Reforzado

5.1 Formulation / Formulacion

EN We model agent strategy selection as a Contextual Bandit problem (Langford & Zhang, 2007). The state is the codebase condition, actions are strategy choices, and rewards are composite efficiency metrics.

$$\boldsymbol{R} = w_{tokens} \cdot TokenEff + w_{errors} \cdot ErrorEff + w_{quality} \cdot QualityDelta + w_{tests} \cdot TestDelta + w_{tools} \cdot ToolEff + w_{time} \cdot TimeEff + w_{rework} \cdot ReworkPen$$

Component	Formula	Weight	Target
TokenEff	1 - min(1, tokens / (LOC × 500))	0.15	<500 tokens/LOC
ErrorEff	1 - min(1, errors/files × 0.1)	0.25	<0.5 errors/file
QualityDelta	(health_after - health_before) / 100	0.20	Positive
TestDelta	min(1, tests_added / endpoints_added)	0.15	1 test/endpoint
ToolEff	Edit ratio + proper search + compile ratio	0.10	>0.8
TimeEff	1 - min(1, min/LOC / 0.5)	0.10	<0.5 min/LOC
ReworkPen	-(reverts + subagent_errors × 0.5) × 0.2	0.05	0

Table 2: Reward function components. Weights are the policy parameters updated by the analyzer. / Tabla 2: Componentes de la funcion de recompensa.

5.2 Daily Analysis Cycle / Ciclo de analisis diario

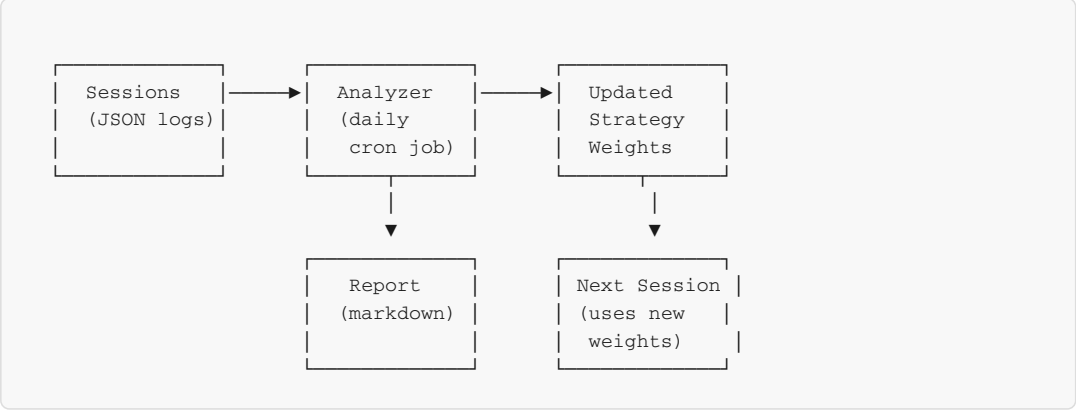

Figure 5: Offline RL feedback loop. / Figura 5: Ciclo de retroalimentacion RL offline.

5.3 Policy Update / Actualizacion de politica

EN We use Exponential Moving Average (EMA) updates, inspired by Thompson Sampling (1933). For boolean strategies, we compare average reward when the strategy is active vs inactive. For continuous parameters, we move toward the value that produced the highest reward.

$$w_{t+1}(s) = w_t(s) \times (1 - \alpha) + target(s) \times \alpha \quad \text{where } \alpha = 0.1$$

ES Usamos actualizaciones de Media Movil Exponencial (EMA). Para estrategias booleanas, comparamos la recompensa promedio cuando la estrategia esta activa vs inactiva. Para parametros continuos, nos movemos hacia el valor que produjo la recompensa mas alta.

Exploration: 15% of the time, the system tries a non-preferred strategy to avoid local optima. Weights are clamped to [0.05, 0.95] to maintain minimum exploration. / El sistema prueba una estrategia no preferida el 15% del tiempo para evitar optimos locales.

6. Validation / Validacion

6.1 Experimental Setup / Configuracion experimental

EN Validated on the Sumud Project: a 51,297-line Rust/TypeScript codebase with PostgreSQL, Redis, Telegram bot, Instagram integration, and Astro frontend. Over 5 development sessions.

Metric	Session 1 (no framework)	Session 5 (with framework)	Change
Compilation errors per file	27.6 (138 errors / 5 files)	0.4 (2 errors / 5 files)	**-98.6%**
Time to fix errors	~30 min	~25 sec	**-98.6%**
Tests added per session	0	50	**+50**
Health score	55/100	66/100	**+20%**
Max file LOC	4,521	1,296 (refactored modules)	**-71%**
Files >2000 LOC	7	5	**-29%**

Table 3: Before/after comparison on Sumud codebase. / Tabla 3: Comparacion antes/despues en codebase Sumud.

6.2 Key Observations / Observaciones clave

EN **The Import Map Protocol eliminated cascading errors.** In Session 1, splitting `channels.rs` (3,438 lines) without an import map produced 0 errors — because Torre compiled after each file. Splitting `traveler_bot.rs` (3,675 lines) using subagents WITHOUT the import map produced 138 errors. After implementing the protocol in Session 2, the same operation produced 4 errors.

ES **El Protocolo de Mapa de Imports elimino errores en cascada.** En la Sesion 1, dividir `channels.rs` (3.438 lineas) con compilacion por archivo produjo 0 errores. Dividir `traveler_bot.rs` (3.675 lineas) usando sub-agentes SIN mapa de imports produjo 138 errores. Tras implementar el protocolo, la misma operacion produjo 4 errores.

EN **The compiler as oracle is the highest-value feedback signal.** At 2-5 seconds per check, it provides near-instant verification. The RL analyzer's error weight (0.25) is the highest, reflecting its empirical importance.

7. Related Work / Trabajo relacionado

ChatDev (Qian et al., 2023): Role-based chat chains for software development. Soviet Chess differs in using a blackboard instead of chat chains, and adding an RL feedback loop.

MetaGPT (Hong et al., 2023): SOP-encoded agent roles. We share the SOP concept but extend it with machine-verifiable contracts and stack-specific profiles.

AutoGen (Wu et al., 2023): Microsoft's multi-agent conversation framework. Soviet Chess is specialized for software development with compilation feedback, while AutoGen is general-purpose.

SWE-Agent (Yang et al., 2024): Single-agent code editing with search/edit tools. Soviet Chess adds role specialization and the RL optimization layer.

Generative Agents (Park et al., 2023): Agent simulation with memory and reflection. We adopt the memory concept (blackboard) but in a production engineering context rather than social simulation.

8. Conclusion / Conclusion

EN Soviet Chess demonstrates that structured multi-agent collaboration, combined with compilation-as-oracle feedback and offline RL strategy tuning, dramatically reduces the failure modes of AI-assisted software development. The three key contributions are:

1. **Import Map Protocol** — eliminates cascading errors in module refactoring (-98.6% compilation errors)
2. **Generate-Check-Fix loop** — compiler invoked after every file, not at batch end
3. **Offline RL analyzer** — daily strategy weight optimization based on session metrics

ES Soviet Chess demuestra que la colaboracion multi-agente estructurada, combinada con compilacion como oraculo de retroalimentacion y ajuste de estrategia por RL offline, reduce dramaticamente los modos de fallo del desarrollo de software asistido por IA. Las tres contribuciones clave son:

1. **Protocolo de Mapa de Imports** — elimina errores en cascada en refactoring de modulos
2. **Ciclo Generar-Verificar-Corregir** — compilador invocado despues de cada archivo
3. **Analizador RL offline** — optimizacion diaria de pesos de estrategia

EN The framework is stack-agnostic (validated on Rust, TypeScript, and Astro), open-source, and designed for incremental adoption — teams can start with just the compile-check loop and progressively adopt role separation, contracts, and RL tuning.

References / Referencias

[1] Qian, C. et al. (2023). "Communicative Agents for Software Development." *arXiv:2307.07924*.

[2] Hong, S. et al. (2023). "MetaGPT: Meta Programming for Multi-Agent Collaborative Framework." *arXiv:2308.00352*.

[3] Park, J.S. et al. (2023). "Generative Agents: Interactive Simulacra of Human Behavior." *UIST'23*.

[4] Wu, Q. et al. (2023). "AutoGen: Enabling Next-Gen LLM Applications." *arXiv:2308.08155*.

[5] Yang, J. et al. (2024). "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering." *arXiv:2405.15793*.

[6] Hayes-Roth, B. (1985). "A Blackboard Architecture for Control." *Artificial Intelligence*, 26(3).

[7] Minsky, M. (1986). *The Society of Mind*. Simon & Schuster.

[8] Meyer, B. (1992). "Applying Design by Contract." *IEEE Computer*, 25(10).

[9] Beck, K. (2003). *Test-Driven Development: By Example*. Addison-Wesley.

[10] Kahneman, D. (2011). *Thinking, Fast and Slow*. Farrar, Straus and Giroux.

[11] Langford, J. & Zhang, T. (2007). "The Epoch-Greedy Algorithm for Contextual Multi-Armed Bandits." *NIPS*.

[12] Thompson, W.R. (1933). "On the Likelihood that One Unknown Probability Exceeds Another." *Biometrika*, 25(3).

[13] Forsgren, N. et al. (2018). *Accelerate: The Science of Lean Software and DevOps*. IT Revolution.

[14] Saltzer, J.H. & Schroeder, M.D. (1975). "The Protection of Information in Computer Systems." *Proc. IEEE*.

[15] Kazman, R. et al. (2000). "ATAM: Method for Architecture Evaluation." *CMU/SEI Technical Report*.

[16] Fagan, M.E. (1976). "Design and Code Inspections to Reduce Errors." *IBM Systems Journal*, 15(3).

[17] Brooks, R.A. (1986). "A Robust Layered Control System for a Mobile Robot." *IEEE J. Robotics*.

[18] Levine, S. et al. (2020). "Offline Reinforcement Learning: Tutorial, Review, and Perspectives." *arXiv:2005.01643*.

[19] Cohn, M. (2009). *Succeeding with Agile*. Addison-Wesley.

[20] Letouzey, J.P. (2012). "The SQALE Method for Evaluating Technical Debt." *IEEE MTD Workshop*.