

CWC

DETERMINISTIC PARALLEL WORK WITHIN ONE BLOCK

Multi-operation atomic execution as a foundation
for future useful-work blockchain architectures

DETERMINISTIC WORK / CANONICAL STATE



CWC / v1

COMPOSE IN PARALLEL. COMMIT CANONICALLY.

A formal treatment of typed conflicts, deterministic scheduling,
bounded execution, state commitments, adversarial controls
and the still-open bridge from work execution to consensus.

Abstract

A conventional block contains many operations, yet deterministically processing its contents can still impose unnecessary sequential execution. The public PoUT CWC execution model explores a sharper abstraction: derive conflicts directly from typed read/write footprints; construct an acyclic precedence relation consistent with an already established candidate order; execute independent operations in real scoped worker threads; merge all effects canonically; and expose a state transition only upon complete candidate acceptance. The research promise is not the mere existence of parallel transactions—a well-established field—but the composability of *bounded deterministic concurrency*, *reproducible state transitions* and *explicit integrity boundaries* in a future useful-service architecture.

We formalize the finite CWC model, state a conditional serial-equivalence argument, derive work/span and contention limits, inspect failure atomicity, present an actual public five-operation example, and distinguish modeled speedup envelopes from measured results. The public source uses synthetic unsigned counters, not real assets, node admission, a live ledger or consensus weight. We conclude with a falsifiable engineering and protocol research agenda rather than a performance or deployment promise.

THE TECHNICAL THESIS. The compelling engineering opportunity is *one ordered candidate, multiple independently executable state transitions, one canonical outcome*. The unsolved scientific opportunity is to compose that executor with independently authenticated useful-service evidence and a secure global admission/finality rule—without converting a scheduler into a fictitious proof of useful resources.

Keywords: deterministic parallel execution; conflict serializability; dependency DAG; bounded state machines; transaction atomicity; canonical encoding; BLAKE3 commitment; useful-service verification; adversarial composition.

Contents

		9 Adversarial tests: proving the tests can fail	9
Abstract	1	10 Threat taxonomy and nonclaims	10
1 The execution thesis: one candidate, many tasks	2	11 Prior art, differentiation and research novelty	10
2 A bounded, executable state-transition machine	2	12 Experimental program: what a real breakthrough must demonstrate	11
3 Conflict geometry: deriving the dependency DAG	3	13 Conclusion: a compelling conditional architecture	12
4 Conditional serial-equivalence argument	4	A Formal invariant ledger	13
5 Atomicity: all the work or none of the state	5	B Pseudocode for conflict construction and barrier execution	13
6 Canonical encoding as a consensus-adjacent primitive	6	C Worked five-operation state trace	14
7 What parallelism could buy: math before marketing	6	D Reproducibility checklist and source boundaries	14
8 Why multi-operation composition matters beyond counters	8	E References and research provenance	14

1 The execution thesis: one candidate, many tasks

1.1 What makes the idea compelling

The elementary observation is structural. If two operations modify independent pieces of state, executing one only after the other is unnecessary for correctness. With many independent operations, a deterministic system may have significant *latent parallel width*. The challenge is preserving a single externally reproducible result while admitting physical concurrency and retaining intelligible rejection semantics.

CWC defines a narrow solution for an *already ordered candidate*. Let

$$B = (o_0, o_1, \dots, o_{n-1}), \quad n \leq 256, \quad (1)$$

where each o_i is a finite, typed operation on an existing key/value state S_0 . The source order is normative. A correct parallel executor may schedule independent operations concurrently but must not alter the meaning of that order.

$$\underbrace{\text{Exec}_{\text{parallel}}(S_0, B, p)}_{\text{actual concurrent workers}} \stackrel{\text{accepted}}{\equiv} \underbrace{\text{Exec}_{\text{serial}}(S_0, B)}_{\text{ordered semantic reference}}, \quad p \in \{1, 2, 4\}. \quad (2)$$

The equivalence concerns the *entire* accepted result: final state, operation results, canonical encodings and 256-bit state commitment. Local aborts are explicitly not evidence of deterministic candidate invalidity.



Figure 1: Public CWC architecture: order enters as input; typed accesses determine the graph; workers execute behind barriers; the final commitment is canonical. This is an executor, not a live consensus network.

1.2 The real innovation question

Parallel transaction processing is not new: Solana’s Sealevel supports nonconflicting account access, and Aptos Block-STM uses speculative execution and validation against a fixed order [4, 5, 6]. A defensible PoUT innovation claim must therefore concern a more specific *system composition*: transparent conflict derivation, deterministic multi-operation results, bounded resource semantics, auditable rejection, and a future interface to useful-service validation. The last interface is a research requirement, *not* an integrated public implementation.

2 A bounded, executable state-transition machine

2.1 Typed keys, values and noncreation semantics

Let the finite key set be $K \subseteq \{0, 1\}^{256}$, with $|K| \leq 1024$. Existing values are 64-bit unsigned integers:

$$S : K \longrightarrow \{0, 1, \dots, 2^{64} - 1\}; \quad \text{dom}(S) = K. \quad (3)$$

A missing key differs from an existing key whose value is zero. State keys are represented by literal 32-byte strings and sorted lexicographically in the canonical representation. Version 1 has four operation types:

Typed operation	$\mathcal{R}(o)$	$\mathcal{W}(o)$	Semantic obligation
$\text{AssertEquals}(k, v)$	$\{k\}$	$\emptyset$	Require existing $S(k) = v$; record observed value.
$\text{Set}(k, v)$	$\emptyset$	$\{k\}$	Replace value at an existing key; never create a new key.
$\text{Add}(k, a)$	$\{k\}$	$\{k\}$	Checked $S(k) + a$; overflow rejects candidate.
$\text{Move}(a, b, x)$	$\{a, b\}$	$\{a, b\}$	Distinct existing keys; checked debit and credit.

This type system is deliberately austere. There is no floating-point nondeterminism, no implicit state-key creation, no network access in the evaluator, and no real token transfer capability. Set is not a minting authorization. Everything is synthetic and in-memory.

2.2 Accepted, rejected, aborted are separate propositions

The public API distinguishes three outcomes:

$$\text{Outcome} = \text{Accepted}(S_f, \rho_f, r) \mid \text{Rejected} \mid \text{Aborted}(e_{\text{local}}). \quad (4)$$

A failed assertion or checked arithmetic violation invalidates the candidate. A thread-spawn failure, local allocation failure, cancellation or worker panic makes the attempt incomplete. It does not prove that the candidate itself is invalid. This distinction becomes fundamental when executions may be retried on heterogeneous nodes.

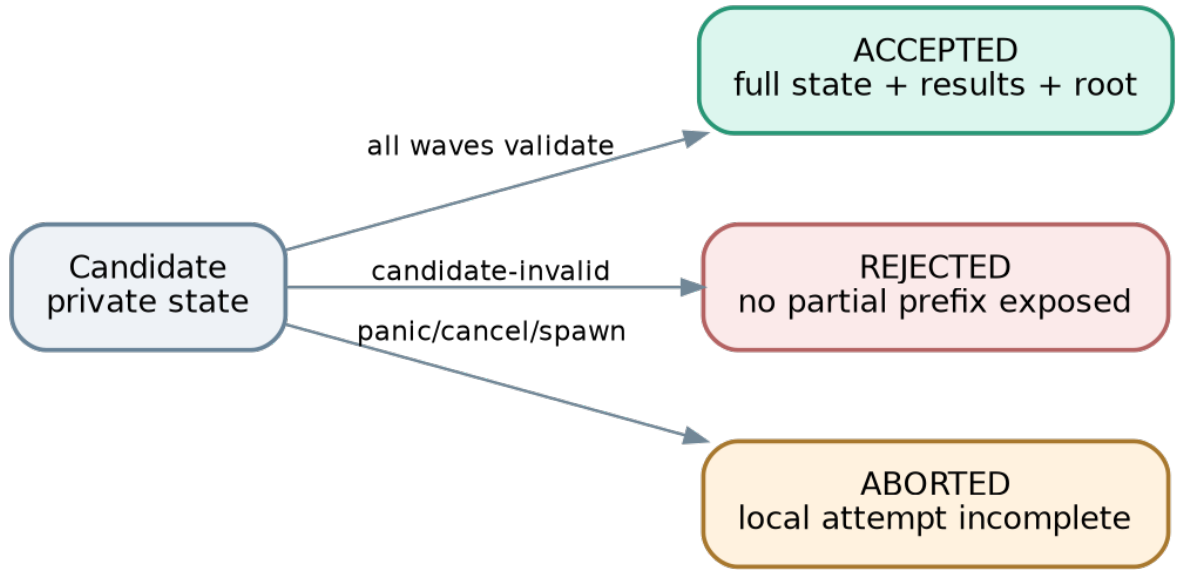


Figure 2: Typed outcome lattice. Only full success publishes a new candidate state; a local abort must never be silently converted to global invalidity.

3 Conflict geometry: deriving the dependency DAG

3.1 Read-after-write, write-after-read, write-after-write

For each ordered pair $i < j$, form a conflict edge $o_i \rightarrow o_j$ if any of three dependence classes is present:

$$\text{RAW}(i, j) \iff \mathcal{W}(o_i) \cap \mathcal{R}(o_j) \neq \emptyset, \quad (5)$$

$$\text{WAW}(i, j) \iff \mathcal{W}(o_i) \cap \mathcal{W}(o_j) \neq \emptyset, \quad (6)$$

$$\text{WAR}(i, j) \iff \mathcal{R}(o_i) \cap \mathcal{W}(o_j) \neq \emptyset. \quad (7)$$

Consequently,

$$(i, j) \in E \iff i < j \wedge [\text{RAW}(i, j) \vee \text{WAW}(i, j) \vee \text{WAR}(i, j)]. \quad (8)$$

Because every edge points from lower to higher candidate index, $G = (V, E)$ is acyclic. Read/read overlap alone does not require serialization. The graph is derived from the actual typed operations; callers cannot supply a weaker substitute.

The conservative graph is potentially dense. With at most 256 operations, the edge ceiling is exactly

$$|E| \leq \binom{256}{2} = 32,640. \quad (9)$$

The public executor uses a fixed predecessor bitset with 8,192 bytes, which bounds graph bookkeeping independently of the magnitude of input values. A conservative all-pairs scan is $O(n^2 a^2)$ if a denotes maximal operation footprint size; here typed footprints have at most two keys in each set.

3.2 Depth waves are derived, not voted on

Define the conflict depth recursively:

$$d(j) = \begin{cases} 0, & \text{pred}(j) = \emptyset, \\ 1 + \max_{i:(i,j) \in E} d(i), & \text{otherwise.} \end{cases} \quad \Omega_k = \{o_j : d(j) = k\}. \quad (10)$$

All operations in a wave Ω_k are pairwise nonconflicting. They read the same immutable snapshot taken after wave $k - 1$ and compute isolated results. Their writes are then merged in canonical operation-index order. A barrier separates waves.

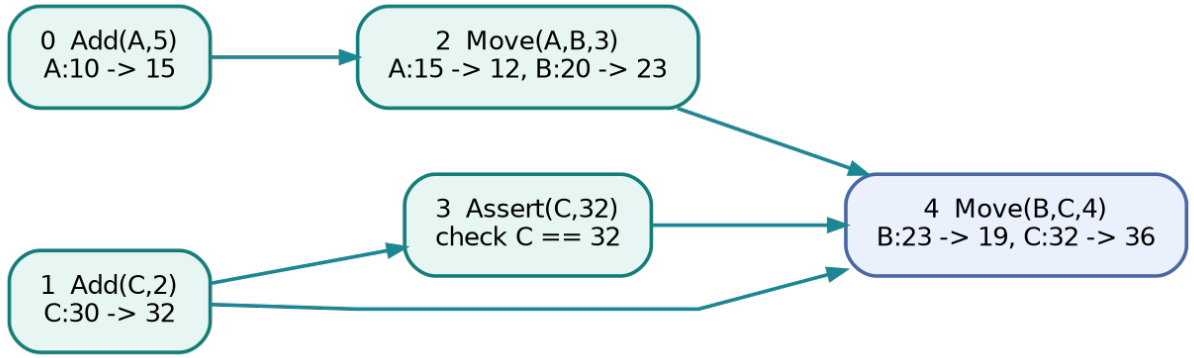


Figure 3: Exact public five-operation example on counters A, B, C . Waves are $\Omega_0 = \{0, 1\}$, $\Omega_1 = \{2, 3\}$, $\Omega_2 = \{4\}$. Dependency edges are derived from typed read/write sets, not invented by a caller.

4 Conditional serial-equivalence argument

4.1 A commutativity lemma

Lemma 1 (nonconflicting footprint commute). Suppose two operations o_i, o_j are defined and valid on the same accessible state, and

$$\mathcal{W}(o_i) \cap (\mathcal{R}(o_j) \cup \mathcal{W}(o_j)) = \emptyset, \quad \mathcal{R}(o_i) \cap \mathcal{W}(o_j) = \emptyset. \quad (11)$$

Then their read values are invariant under exchanging their execution order, their write sets are disjoint, and their state effects commute.

Argument. No operation writes a key that the other reads or writes. Thus each reads the same input values whichever runs first. Their outputs depend only on those reads and the typed constants, and their disjoint writes can be merged in either order. The claim relies on the typed evaluator's absence of additional hidden mutable effects. $\square$

4.2 Wave induction and canonical evidence

Proposition 2 (accepted trace equivalence, model scoped). Assume the typed access derivation is sound, each wave evaluates against one immutable snapshot, every worker finishes, all candidate checks succeed, and effects are merged in index order. Then the final parallel state equals the candidate-order serial state. If result records are normalized by original operation index, canonical result arrays and state roots also agree.

Proof sketch. Induct over increasing conflict depth. Each wave has no internal conflicts and can be exchanged into its original candidate-order placement using Lemma 1 without modifying any read

value or write. Every cross-wave conflicting predecessor belongs to an earlier wave; barriers ensure its effects are present. Thus wave k extends the same serial-prefix state as the reference. Canonical result ordering and deterministic encoding then imply identical root bytes. $\square$

SCOPE OF THE CLAIM. This is a proof sketch for the defined typed state machine and successful local execution, not a mathematical proof that an arbitrary network, malicious host, external oracle or resource-constrained adversary will produce the same trace. External nondeterminism, concurrency admission and durable process-crash recovery are outside the public prototype.

4.3 Why canonical ordering matters

Parallel completion order is deliberately not the consensus-visible result order. For independent operations o_0 and o_2 , a worker may finish o_2 first; accepting that completion order as result order would create machine-dependent encodings. CWC indexes results by their original candidate positions. Hence the intended invariant is

$$\text{Serialize}(\rho_{\text{parallel}}) = \text{Serialize}(\rho_{\text{serial}}), \quad \text{Root}(S_f^{\text{parallel}}) = \text{Root}(S_f^{\text{serial}}). \quad (12)$$

5 Atomicity: all the work or none of the state

5.1 Private candidate state

CWC holds a private candidate copy during evaluation. Accepted results expose an entire successor state, operation results and a root; invalid candidates reveal no committed prefix. For a successful candidate B the public transition interface is

$$\text{publish}(S_0, B) = \begin{cases} (S_f, \rho_f, \text{Root}(S_f)), & \text{if every wave succeeds,} \\ \emptyset, & \text{if rejected or locally aborted.} \end{cases} \quad (13)$$

This is *in-memory atomic candidate publication*. It is not database durability and does not confer global transaction finality.

5.2 Adversarial shared-reserve example

Let $S_0(R) = 100$, $S_0(U) = S_0(V) = 0$ and let the candidate contain

$$B = [\text{Move}(R, U, 80), \text{Move}(R, V, 80)]. \quad (14)$$

The pair conflicts on the shared reserve R : the first move would leave $R = 20$, making the second debit invalid. It is unsound to execute both against the untouched $R = 100$ snapshot and commit their credits. Correct derived dependencies serialize them, and the candidate is rejected in its entirety; S_0 remains untouched. This is a synthetic overdraw invariant, not a token-transfer test.

Listing 1: Public API shape, adapted from the released synthetic example.

```

1 let state = State::new(vec![(a, 10), (b, 20)]?);
2 let block = Block::new(vec![
3     Operation::Add { key: a, amount: 5 },
4     Operation::Add { key: b, amount: 7 },
5     Operation::Move { source: a, destination: b, amount: 3 },
6 ])?;
7 let reference = execute_serial(&state, &block);
8 let parallel = execute_parallel(&state, &block, 4, &cancel);
9 assert_eq!(reference, parallel);
10 // Accepted final values: A = 12, B = 30.
```

5.3 Failure containment is a core requirement

A locally failing worker is joined before the attempt returns. A cancellation flag cannot become an admission verdict. Spawn limits and memory failures are not candidate failures. Typed outcome separation is essential for reproducible retries and future distributed diagnostics. No claim is made for force-kill or host-crash conditions that prevent in-memory cleanup.

6 Canonical encoding as a consensus-adjacent primitive

6.1 Committed meaning, not completion timing

CWC encodes versioned keys, unsigned values, operations and results using fixed-width little-endian integers and literal key bytes. A canonical state is ordered by 32-byte key, with unknown tags, duplicate keys, truncation and trailing bytes rejected. In the public v1 model,

$$\text{Root}(S) = \text{BLAKE3}_{256}(\text{POUT/CWC/STATE_ROOT} \parallel 0 \times 00 \parallel \text{LE16}(1) \parallel \text{LE32}(|\sigma(S)|) \parallel \sigma(S)), \quad (15)$$

where $\sigma(S)$ is the canonical encoded state, not a generic language-object serialization.

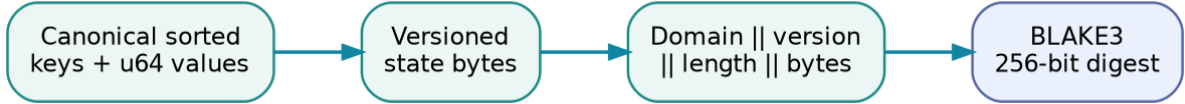


Figure 4: Domain-separated state commitment. Matching roots are necessary integrity evidence for this synthetic state; they do not authenticate transaction legitimacy, a blockchain ancestor, honest setup or economic admission.

6.2 Finite codec boundaries

Encoded record	Input cap (bytes)	Max constructible (bytes)
Canonical state	41,000	40,979
Ordered candidate	65,536	18,707
Operation results	21,300	21,269
Whole execution	65,536	62,309

The finite limits are not optimization trivia: they make input-driven allocation, size rejection, parsing consistency and testing tractable. Maximum state keys: 1,024. Maximum candidate operations: 256. Actual worker configurations: 1, 2 or 4. Maximal thread spawn requests: 256 per candidate. Requested worker stack: 256 KiB per thread, excluding platform overhead.

7 What parallelism could buy: math before marketing

7.1 Work, span and execution limits

Let t_i be the hypothetical cost of operation i , $W = \sum_{i=0}^{n-1} t_i$ total work, L the maximum directed-path work (span), and p the number of available workers. Any parallel schedule has

$$T_p \geq \max\left\{\frac{W}{p}, L\right\}, \quad S_p = \frac{T_1}{T_p} \leq \min\left\{p, \frac{W}{L}\right\}. \quad (16)$$

The current wave-barrier implementation also imposes depth synchronization. For equal unit-cost operations and negligible scheduling overhead, a simple layered capacity model is

$$T_{p,\text{wave}}^{\text{ideal}} = \sum_{k=0}^{D-1} \left\lceil \frac{|\Omega_k|}{p} \right\rceil, \quad S_{p,\text{wave}}^{\text{ideal}} = \frac{n}{T_{p,\text{wave}}^{\text{ideal}}}. \quad (17)$$

In practice, thread spawn, snapshot cloning, canonical merge, synchronization, cache effects and memory allocation contribute additional nonnegative overhead:

$$T_p^{\text{actual}} = T_p^{\text{compute}} + T^{\text{schedule}} + T^{\text{barrier}} + T^{\text{merge}} + T^{\text{memory}}. \quad (18)$$

The public research release *does not* report a measured speedup; tiny synthetic counter operations can be slower with worker threads. The following figures therefore show *mathematical scenarios*, never claimed PoUT measurements.

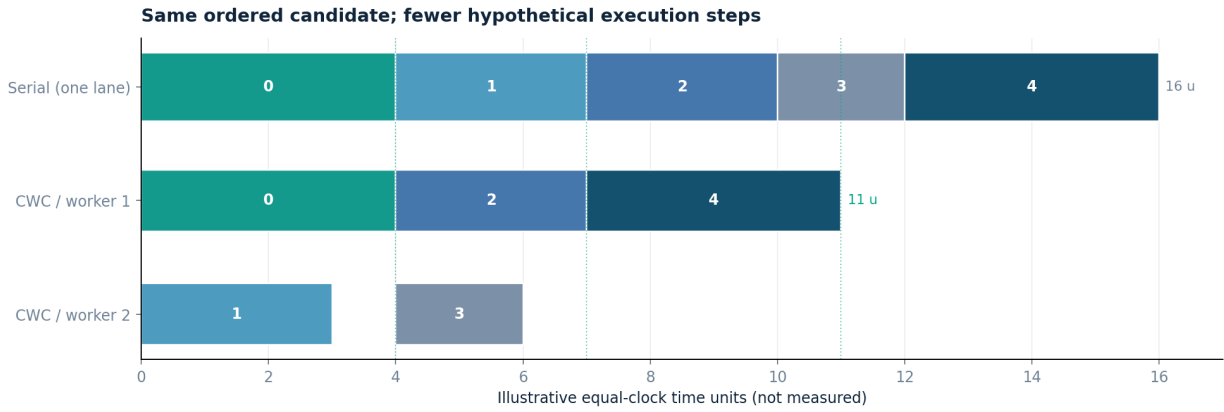


Figure 5: Illustrative five-operation work schedule with explicitly invented time units $t = (4, 3, 3, 2, 4)$. Serial work is 16 units; an idealized two-lane barrier schedule takes 11 units (1.45×). No throughput, wall-clock or deployed-system claim follows.

7.2 Amdahl and the seductive trap of theoretical speedup

With perfectly parallelizable fraction α and ideal p -way parallel execution of only that fraction,

$$S(p; \alpha) = \frac{1}{(1 - \alpha) + \alpha/p}. \quad (19)$$

Amdahl's law is a scenario-level bound, not a characterization of the CWC public workload. A highly parallel candidate may still incur enough graph-construction and thread costs to lose wall-clock performance.

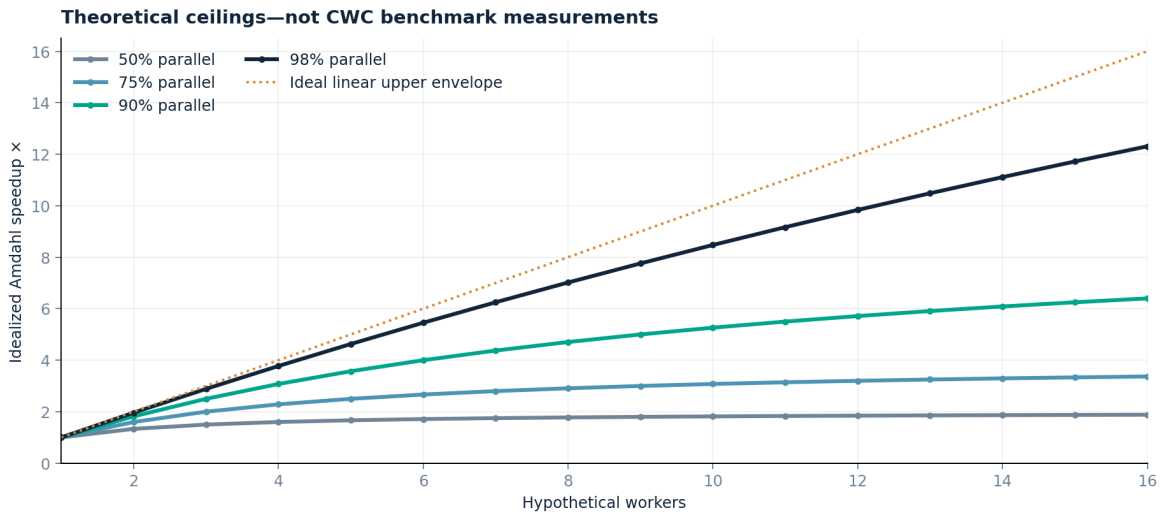


Figure 6: Analytical Amdahl envelopes for hypothesized parallel fractions. None of these lines is an observed PoUT benchmark; the public executor permits only 1, 2 or 4 configured workers.

7.3 Key contention reduces graph width

Consider an *invented* 64-operation workload in which every operation writes exactly one randomly selected key. As the set of possible keys shrinks, more operations compete for the same key and depth layers grow. Figure 7 reports a fixed-seed *scheduler-only* abstraction averaging 40 toy instances per setting. It uses ideal unit task costs and ignores actual Rust runtime overhead.

7.4 A throughput claim must survive the whole execution envelope

A credible future measurement would report distributions over T_{graph} , T_{workers} , T_{merge} , T_{encode} and T_{total} ; absolute p50/p95 latency; admission and memory overhead; exact operation mixes; conflict patterns;

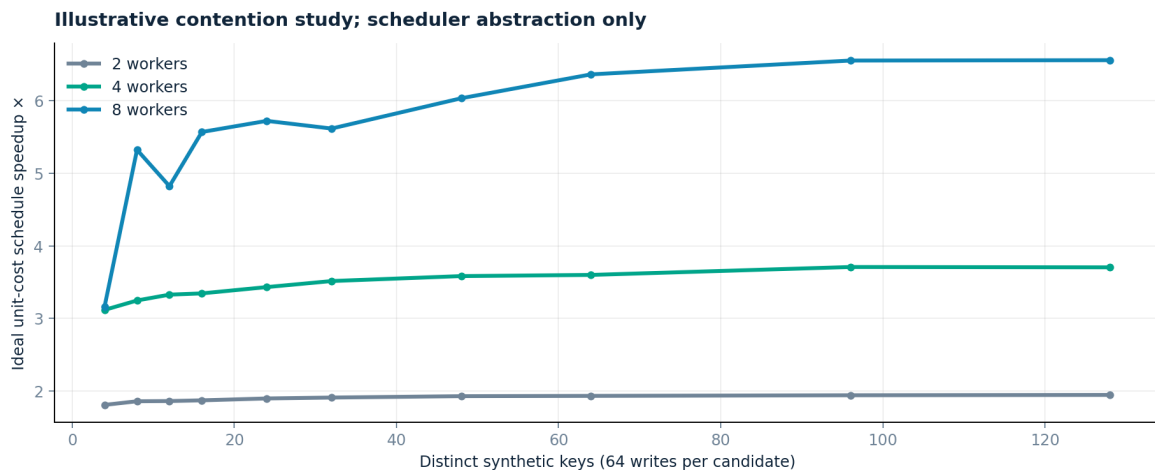


Figure 7: Synthetic graph-width sensitivity, not a CWC benchmark. The 8-worker curve is an algebraic extrapolation; the public CWC worker API is limited to 1/2/4.

CPU topology; and serial-reference equality. The relevant business- or consensus-level metric must additionally include *external data availability*, *validation cost*, *state storage* and *network effects*. Otherwise impressive thread throughput can hide a slower usable system.

8 Why multi-operation composition matters beyond counters

8.1 The difference between a scheduler and a protocol substrate

Executing two independent adds in one candidate is mathematically modest. The architectural payoff is a deterministic substrate into which well-specified typed operations *could* eventually be admitted. Figure 8 draws the boundary between the existing kernel and separate research obligations.

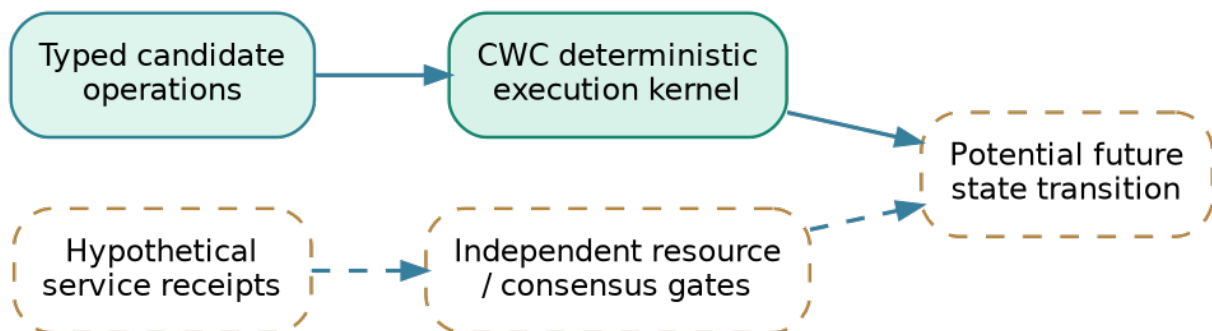


Figure 8: Potential future composition. Dashed boxes are not implemented by the publicly released CWC crate. Service receipt validity, independent demand, secure eligibility and finality require separate mechanisms.

A hypothetical future candidate might need to verify a storage receipt, account for a customer-authorized retrieval, update a usage ledger and apply an independent service-side accounting adjustment. If the state footprints are disjoint, parts can execute concurrently; if they compete for a shared escrow or admission counter, their dependencies enforce ordering. **The interesting feature is one canonical final result despite internal concurrency, not a magical multiplication of work.**

$$B_{\text{future}} = (o_{\text{receipt}}, o_{\text{audit}}, o_{\text{usage}}, o_{\text{reserve}}, \dots), \quad (20)$$

$$\text{ValidCandidate} \Rightarrow \bigwedge_i \text{ValidOp}_i \wedge \text{Serializable}(B_{\text{future}}), \quad (21)$$

$$\text{EligibleService} \not\Rightarrow \text{ValidCandidate} \quad \text{without external demand/resource assumptions.} \quad (22)$$

The third line is the crucial security boundary: a deterministic state-machine result does not establish economically independent, scarce physical work.

8.2 Composable work units and a future independent observer

One can conceptually separate each accepted unit into three obligations:

1. **Semantic integrity.** The output matches the typed computation and state transition.
2. **Service authenticity.** A distinct authenticated party requested and observed the relevant service.
3. **Resource soundness.** A coalition cannot earn arbitrary opportunities from negligible shared backing or freely fabricated observations.

CWC publicly addresses a finite form of obligation 1 for synthetic counters. Obligations 2 and 3 are *open* in the broader PoUT research. A future noncircular consensus composition would further need common admission history, unpredictable challenges, canonical weight, fork reconciliation and finality. None follows from adding threads.

9 Adversarial tests: proving the tests can fail

9.1 The differential-testing equation

The public CWC test approach compares three forms of execution: serial reference, actual parallel executor and an independently written u128 arithmetic oracle for small candidates. The desired finite regression invariant is

$$\forall B \in \mathcal{C}_{\text{tested}} : \text{Outcome}_{\text{oracle}}(B) = \text{Outcome}_{\text{serial}}(B) = \text{Outcome}_{\text{parallel}}^{(p)}(B), \quad p \in \{1, 2, 4\}, \quad (23)$$

where oracle comparison covers validity and accepted state/results/root, and local-abort controls are tested separately. Passing a finite corpus is not a universal theorem; the independence of the oracle's arithmetic implementation is what makes a shared mistake less likely to be hidden.

9.2 Finite public evidence ledger

Public control	Evidence / scope
Finite enumeration	5,265 synthetic candidates from a fixed, small two-key/operation alphabet.
Worker configurations	1, 2 and 4 actual scoped worker configurations compared with serial.
Fault classes	Arithmetic overflow/underflow, missing keys, failed assertions, write aliases and shared-reserve overspend.
Concurrency faults	Worker panic, thread-spawn failure, cancellation and reversed completion.
Encoding negative controls	Unknown versions/tags, duplicates, out-of-order keys, truncation and trailing bytes.
Deliberate dependency defect	Omit a RAW edge in an isolated test and demonstrate serial/parallel disagreement.
Historical test counts	Original release: 21 default crate unit tests + 12 integration tests, plus 5 feature tests; totals overlap across configurations.

The public research report records 41 passing default-workspace tests for the *original* release. Later public addenda, including the byte-observation adaptation, do not inherit an independently executed full Rust pass merely because they retained CWC source. The broader private engineering workspace’s test counts are neither CWC benchmark evidence nor validation of the smaller exported tree.

9.3 Mutation resistance: the missing-edge countermodel

For candidate $[\text{Set}(A, 2), \text{AssertEquals}(A, 2)]$ with $S_0(A) = 1$, an unsound graph with no RAW edge might evaluate both operations on the original snapshot: the assertion incorrectly sees $A = 1$ and fails. The correct graph orders the assertion after the set, matching the serial acceptance. A deliberate missing-edge control is therefore not a decorative negative test; it proves that the test harness can *detect precisely the class of scheduling defect the model is designed to exclude*.

10 Threat taxonomy and nonclaims

Threat or question	Necessary defense / test	Current status
Unmodeled state effects	Typed complete footprint and independent evaluator semantics	Narrow finite model
Conflicting shared reserve	RAW/WAR/WAW ordering, checked arithmetic, total rejection	Public regression
Completion-order nondeterminism	Sort by original candidate index, canonical state encoding	Public regression
Partial accepted prefix	Isolated candidate state; no publish on reject or abort	Public regression
Malformed input	Bounded canonical codec and negative controls	Public regression
Host exhaustion	Bounded workers/spawns, local abort, per-host admission required	Host-level gate open
Crash recovery	Durable journal/replay semantics	Not implemented here
Physical useful-service scarcity	Coalition-wide lower bound with shared state and all assistance	Not established
Decentralized eligibility / weighting	Authenticated history, freshness, demand, fork safety	Not established
End-to-end performance	Exact-source workload/latency/throughput and hardware controls	Not measured

BOUNDARY DECLARATION. The public CWC executor is not a live blockchain, does not implement a GHOSTDAG consensus rule, does not select permissionless validators, does not authenticate third-party demand and cannot claim a proven multi-task speedup. Mathematical diagrams below illustrate conditional capacity or correctness, not investment performance or guaranteed implementation outcomes.

11 Prior art, differentiation and research novelty

11.1 Comparison without a false first-in-the-world claim

System	Core public approach	Relevant distinction
Solana Sealevel	Account access annotations enable concurrent execution of nonconflicting transactions.	Mature chain execution; not evidence of PoUT resource-soundness.
Aptos Block-STM	Optimistic speculative execution with multiversion state and revalidation against fixed order.	Handles dynamic conflicts differently from static typed-wave barriers.
PoUT CWC public v1	Declared typed operations $\rightarrow$ derived dependency DAG $\rightarrow$ barriered worker waves $\rightarrow$ one canonical result.	Narrow bounded synthetic executor; simple and auditable, not throughput-validated.

The technically bullish path is *system-specific composition*: use a simple, auditable deterministic kernel as one building block while separately establishing service observability, adversarial resource costs and consensus safety. That combination is a proposed research direction. Neither the name CWC nor the ability to execute several operations in one block constitutes a demonstrated global invention by itself.

11.2 Three defensible contributions of the public artifact

1. An inspectable typed-conflict graph with canonical operation order and bounded memory/worker constraints.
2. A tangible serial/reference/parallel correctness story, including adversarial missing-edge controls and no-partial-state outcomes.
3. A compact platform for specifying what a later multi-service candidate integration would have to preserve—and which external assumptions it cannot hide.

This is not to diminish the engineering challenge. Determinism across threads is hard, and getting the boundary right is materially more valuable than a glossy but unfalsifiable scalability claim.

12 Experimental program: what a real breakthrough must demonstrate

12.1 Phase A: deterministic correctness at broader scale

Extend operation semantics only with explicit typed footprint declarations, canonical encodings and authoritative serial semantics. Use exhaustive finite cases, structured random differential tests, mutation controls and negative construction cases. Record counterexamples, not only success tallies. Establish whether depth waves remain suitable as richer operations grow.

12.2 Phase B: empirical performance, not illustrative figures

On each exact-source benchmark run, measure microseconds per candidate, CPU cycles, allocation and peak RSS, worker occupancy, graph-depth distribution, conflict fractions and state-root agreement. Compare single-thread and bounded worker configurations under low/high contention, increasing candidate sizes and operating-system profiles. Include one deliberately serial workload, one wholly independent workload and one heterogeneous-cost workload. Report p50/p95, confidence intervals and code/hardware hashes. A credible positive claim requires *positive measured outcomes that survive overhead*.

12.3 Phase C: durable admission and cross-block persistence

Define append-only replay/journal semantics, canonical parent-state commitments, state rollback boundaries and process-crash recovery. Clarify when a candidate's internal accepted state becomes a globally accepted state; CWC alone cannot do this. Separate authenticating an object from the authority to accept it.

12.4 Phase D: independent useful-service verification

Provide a model of client request, provider service, independent observer, challenge timing, complete public view and coalition-global physical resources. Prove or refute a nonvacuous bound over distinct eligible service opportunities. Count shared stores, cached answers, preprocessing, delegated processing and external observers. A valid file digest does not independently establish the source was scarce or that real customer demand existed.

12.5 Phase E: secure composition and network safety

Only after a verifiable opportunity model exists should it be linked to eligibility, weight and any GHOSTDAG-like ordering hypothesis. Explicitly handle Sybil identities, duplicated services, replay, fork-specific consumption, finality, liveness, adversarial scheduling and dependency/nonce manipulation. Test end-to-end with reproducible network fault injection. No schedule or launch date is promised by this list of gates.

13 Conclusion: a compelling conditional architecture

A candidate need not choose between *ordered meaning* and *concurrent computation*. The public PoUT CWC research prototype demonstrates a concrete way to preserve original order through typed conflict derivation, run nonconflicting operations on actual scoped worker threads and expose a canonical full-candidate result only on success. Its narrow operation semantics, bounded executor and negative tests make it a legitimate engineering research artifact.

The high-upside direction is to turn that artifact into a reusable substrate for authenticated multi-service work, while refusing to confuse proof of correct computation with proof of independently valuable physical service. If future PoUT research can establish those missing security and admission layers, deterministic multi-operation composition could become a powerful architectural advantage. *That is a research thesis worth testing, not a conclusion already proven.*

MULTIPLE OPERATIONS. ONE CANONICAL STATE.

Parallel execution without semantic ambiguity.

The next frontier is secure composition with independently verifiable useful work.

A Formal invariant ledger

Invariant	Meaning / falsifiable check
Order preservation	Every graph edge or canonical result refers to original candidate indices. No operation is semantically moved past a conflict.
No read/write alias escape	$i < j$ with any RAW, WAW or WAR overlap implies $i \rightarrow j$.
Wave independence	$i, j \in \Omega_k, i \neq j \Rightarrow$ no typed conflict.
Read snapshot	Every operation in Ω_k reads the same immutable previous-wave snapshot.
State ownership	S_0 borrowed and never mutated; private candidate copy published only on success.
Atomic reject	Any genuine candidate-invalid event implies no accepted prefix or successor state.
Abort provenance	Cancellation, thread/resource failures and worker panic are local aborts, not deterministic rejection.
Result determinism	ρ sorted by original operation index; sorted key/value representations in each canonical record.
Codec determinism	Fixed domain, version, byte lengths and little-endian encoding; unknown/trailing input rejected.
Arithmetic safety	All $u64$ additions/debits/credits checked; missing or aliased keys rejected.
No public consensus shortcut	Accepted synthetic result does not assign a node eligibility or a consensus weight.

B Pseudocode for conflict construction and barrier execution

The following listing is *illustrative pseudocode adapted to the published semantic design*, not a claim of an exact source-file extract.

Listing 2: Conflict-graph derivation; pseudocode.

```

1  for later in 0..operations.len() {
2    for earlier in 0..later {
3      let r0 = reads(operations[earlier]);
4      let w0 = writes(operations[earlier]);
5      let r1 = reads(operations[later]);
6      let w1 = writes(operations[later]);
7      let raw = intersects(w0, r1);
8      let war = intersects(r0, w1);
9      let waw = intersects(w0, w1);
10     if raw || war || waw {
11       add_edge(earlier, later);
12       depth[later] = depth[later].max(depth[earlier] + 1);
13     }
14   }
15 }
```

Listing 3: Conceptual private candidate execution and full-wave barrier; pseudocode.

```

1  let mut candidate = clone_private(initial)?;
2  let mut collected = Vec::new();
3  for wave in depth_waves(&block) {
4    let snapshot = snapshot_immutable(&candidate);
5    let evaluated = run_scoped_workers(&wave, &snapshot)?;
6    if evaluated.any_invalid_candidate() { return Rejected; }
7    if evaluated.any_local_abort() { return Aborted(local_reason); }
8    for item in sorted_by_original_index(evaluated) {
9      apply_disjoint_writes(&mut candidate, item.writes);
10     collected.push(item.result);
11   }
12 }
13 collected.sort_by_key(|r| r.index);
14 Accepted(finalize_with_canonical_root(candidate, collected))
```

The real public implementation has precise worker joining, error propagation, cancellation and allocation rules. The pseudocode captures only the high-level semantic order; it must not replace the real code in a security review.

C Worked five-operation state trace

Start with $S_0(A, B, C) = (10, 20, 30)$ and the candidate

$$\begin{aligned} o_0 &= \text{Add}(A, 5), & o_1 &= \text{Add}(C, 2), \\ o_2 &= \text{Move}(A, B, 3), & o_3 &= \text{AssertEquals}(C, 32), \\ o_4 &= \text{Move}(B, C, 4). \end{aligned}$$

The derived depth waves are $\Omega_0 = \{0, 1\}$, $\Omega_1 = \{2, 3\}$ and $\Omega_2 = \{4\}$. The canonical intermediate state trace is

$$(10, 20, 30) \xrightarrow{\Omega_0} (15, 20, 32) \xrightarrow{\Omega_1} (12, 23, 32) \xrightarrow{\Omega_2} (12, 19, 36).$$

The important detail is not the trivial arithmetic: o_0 and o_1 execute concurrently but share one canonical wave output, while o_2 and o_3 can execute against its immutable snapshot and o_4 sees both completed writes. State-root equality against serial execution is the acceptance criterion in the public example.

D Reproducibility checklist and source boundaries

Use the public repository listed below. On a suitable Linux development environment, the documented commands include:

Listing 4: Commands published by the current public research repository.

```
1 cargo fmt --all -- --check
2 cargo clippy --locked --workspace --all-targets -- -D warnings
3 cargo test --locked --workspace
4 cargo test --locked -p pout-cwc-research --all-features --all-targets
5 cargo run --locked -p pout-cwc-research --example demo
6 python3 -I -B scripts/check_source_manifest.py
```

All numbers described as *executed* in this paper are attributed to the published release's scope and dates. Re-running these commands on the current public tree is a separate task and is not claimed here. The public CWC code and diagrams are reproducible from its available model specification and example; the conceptual future-PoUT architecture is not executable from this package.

E References and research provenance

References

- [1] PoUT Network, *CWC execution model*, public source and documented finite invariants, 21 October 2026. https://github.com/PoUTnetwork/pout-public-source-v0.1.0-2/blob/main/docs/CWC_MODEL.md.
- [2] PoUT Network, *pout-cwc-research*, public Rust source, demo and differential correctness tests. <https://github.com/PoUTnetwork/pout-public-source-v0.1.0-2/tree/main/crates/pout-cwc-research>.
- [3] PoUT Network, *Public research status and validation scope*. https://github.com/PoUTnetwork/pout-public-source-v0.1.0-2/blob/main/docs/RESEARCH_STATUS.md.
- [4] Solana Foundation, *Solana transactions and writable account access*, public developer documentation. <https://github.com/solana-foundation/developer-content/blob/main/docs/core/transactions.md>.
- [5] R. Gelashvili et al., *Block-STM: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing*, 2022/2023. <https://arxiv.org/abs/2203.06871>.
- [6] Aptos Labs, *Aptos whitepaper: Parallel execution and block pipelining*. <https://github.com/aptos-labs/whitepaper/blob/main/english/main.tex>.
- [7] G. M. Amdahl, *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*, AFIPS, 1967.