Supplementary Material

Synchronous Forward Propagation of Commits in N parallel Git Branches

---

This supplementary document accompanies the main paper of the same title. It carries the full implementation detail, the message templates, the complete event schema, the gap-analysis procedure, the implementation roadmap, and the appendices. Readers consulting the main paper for the framework overview, the workload equation, the production results, and the dashboards do not need this document. Readers building the framework or replicating the analytics pipeline do.

S1. Methodology Critique Detail

The main paper Section 3.1 summarises four manual synchronous propagation approaches in one table. The detail behind each row follows.

S1.1 Blind Synchronous Propagation

Every commit to one branch is replicated to every other active branch. The rule is independent of the product line and the customer. Every codebase has at least one pair of branches with this relationship, usually a customer branch to main. Developers handle this by doing double, triple, or quadruple deliveries. In hectic schedules, developers forget. The forgotten commit becomes a missed feature in the next release.

S1.2 Customer-Based Synchronous Propagation

A feature or bug fix needs to land in a specific subset of future releases. The team decides which branches receive it. Done by hand, this approach hinges on developer due diligence. The operations team carries a manual decision table for every change. Branch coverage drifts over time as people leave the team and tribal knowledge leaks out with them.

S1.3 Milestone-Based Big-Bang Merge

Branches merge into main at the end of a release cycle, all at once. The merge is large. The merge is late. Six concrete failure modes follow.

- **Increased complexity.** The larger the merge, the more complex the conflicts. Conflicts span many files and involve logic harder to untangle.
- **Difficult resolution.** Resolving conflicts after a big-bang merge needs a deep understanding of changes made by multiple developers over weeks or months. The resolution work sometimes loses functionality.
- **Lack of early detection.** Problems and conflicts caught early through regular integration only surface at the end.
- **System breakage.** The main branch is unstable for days or weeks after a big-bang merge.
- **Inadequate testing.** Large merges do not test incrementally.

- **Regression bugs.** New bugs appear. Existing functionality breaks.

In one operational study the main branch stayed in a failure mode for at least two weeks after every release.

S1.4 Rebase as a Strategy

Rebase rewrites commit history by replaying commits on top of an updated base branch. In a semantic versioning environment with parallel major, minor, and patch streams, rebase introduces five operational problems.

- **Loss of history.** Rebasing rewrites the commit graph. Tracing when a feature was introduced becomes harder.
- **Difficulty in auditing.** Reviewers and compliance teams lose the chronological record.
- **Coordination overhead.** Rebase needs constant communication among team members.
- **Reversions.** A bad rebase is harder to revert than a merge.
- **Coordination across many branches.** Inconsistency leads to drift between branches.

S2. Intelligent Follow-Up System

The most overlooked aspect of synchronous propagation automation is sustained developer engagement after the system has done its work. The follow-up system solves this with five mechanisms working together.

S2.1 Multi-Channel Notifications

Each notification picks a channel based on urgency.

Urgency	Channel	Timing	Content
Immediate	Chat direct message	Real-time	One-line summary, conflict probability, three action buttons
High	Email batched	Hourly	Summary of open items requiring action
Medium	Dashboard badge	Daily digest	Weekly synchronous propagation health summary
Low	Weekly report	Monday morning	Trends, team metrics, no immediate action

Smart batching groups notifications by repository, suppresses duplicates, and respects working hours. A chat message at 11 p.m. local time gets deferred to 9 a.m. the next day.

S2.2 Priority Tiers

Every open synchronous propagation belongs to one of four tiers.

- **Tier 1 critical, 0 to 4 hours.** Security fixes, production blockers, cascading dependencies.
- **Tier 2 urgent, 4 to 24 hours.** Merge conflicts, review-ready changes, items approaching SLA.

- **Tier 3 standard, 24 to 72 hours.** Standard reviews and pending developer action.
- **Tier 4 deferred, beyond 72 hours.** Work in progress and blocked dependencies.

S2.3 Automated Escalation Ladder

Stage	Time	Action	Recipients
1	0 hours	Initial notification	Developer
2	4 hours	Reminder, priority bump	Developer + chat DM
3	24 hours	Manager notification	Team Lead added
4	48 hours	Repository owner alert	Repo Owner added
5	72 hours	Executive visibility	Engineering Manager added
6	7 days	Auto-abandon consideration	Process Owner added

The ladder respects weekends, holidays, and on-call rotations.

S2.4 Developer Dashboard

Each developer sees five action lanes. Needs immediate action (red), ready to merge (yellow), in progress (blue), delegated (white), and personal stats. Personal stats include weekly velocity, percentile rank, average review time, and conflict rate.

S2.5 Behavioural Nudges

- Progress visualisation. “Two merges away from clearing this week’s queue.”
- Effort estimation. “This review typically takes five minutes.”
- Social proof. “Three teammates merged similar changes today.”
- Loss aversion. “Will auto-abandon in three days if no action.”

S2.6 Auto-Merge Versus Human Review

The main paper Section 4.4 introduces the four configurable auto-merge policies. The repercussions are spelled out below.

- **Speed versus safety.** `auto_merge_all` ships fixes in seconds and ships bugs in seconds. `require_review_all` ships nothing without a human in the loop.
- **Reviewer load.** Tightening the policy pushes more pull requests into the human review queue. A flip from `auto_merge_low_and_medium` to `require_review_all` raises review load 4x to 6x overnight.
- **Audit posture.** Some compliance regimes require a human approval on every change touching a regulated branch. The framework records the policy in effect at the time of each merge.
- **Trust calibration.** Teams new to the framework start with `require_review_all` and loosen the policy as confidence grows. Dashboards expose the regression rate per policy tier so the choice is data-driven.

- **Reversibility.** Any change to the policy applies forward, not backward.

S2.7 Effectiveness Metrics

Metric	Target
Time to first action	< 4 hours
Notification conversion rate	> 70 percent
Escalation rate	< 15 percent
Developer satisfaction	> 4.0 / 5
Alert fatigue index	< 2 notifications per developer per day
Auto-merge regression rate per tier	< 0.5 percent per quarter

---

S3. Data Mining and Analytics Pipeline Detail

S3.1 Polyglot Persistence

The pipeline writes events to three operational tiers in parallel and asynchronously replicates to a warehouse.

Layer	Technology	Use case	Retention
Relational	MySQL or PostgreSQL	Current state, transactional queries, joins	Permanent
Time-series	InfluxDB or Prometheus	Metric trends, alerting, rollups	2 years
Document store	Elasticsearch or MongoDB	Full-text search, audit logs	5 years
Data warehouse	Snowflake or BigQuery	Long-term analytics, ML training	7+ years

S3.2 Extraction Dimensions

Six entity dimensions are captured: commit, repository, branch, developer, ticket, release. Six event dimensions are captured: synchronous propagation initiated, review requested, conflict detected, approval given, merge completed, failure occurred.

Five derived metrics are computed: synchronous propagation lead time, review wait time, conflict resolution time, idle queue time, developer responsiveness.

S3.3 Four-Tier Analytics

- **Descriptive.** Volume, success rate trends, time-in-state bottlenecks, top contributor leaderboards, failure Pareto.

- **Diagnostic.** Conflict frequency correlated with branch age, file coupling, developer familiarity, commit size. SLA breach analysis by branch pair, time of day, and team.
- **Predictive.** Conflict probability model with five inputs: branch divergence days, files-changed overlap with target, historical conflict rate for files, commit size LOC, developer familiarity score. Synchronous propagation time estimation. Capacity forecasting.
- **Prescriptive.** Recommended reviewer engine based on code ownership and current workload. Best-merge-window suggestion based on historical conflict rates by hour of day. Risk mitigation suggestions before commit.

S3.4 Query Performance Targets

Query type	Latency target	Optimisation
Developer backlog	< 200 ms	Indexed by developer + status
90-day trends	< 2 seconds	Pre-aggregated time-series
Cross-release comparison	< 5 seconds	Materialised views
Full audit trail	< 1 second	Document store with indexes

Aggregation strategy is layered. Last 24 hours stays in raw events. Last 30 days uses hourly rollups. Last 12 months uses daily rollups. Beyond 12 months uses monthly rollups in the warehouse.

S4. Failure Mode Resilience

S4.1 Failure Taxonomy

Failure mode	Detection	Response	Escalation
Merge conflict	CI failure	Developer notification	> 48 h, Manager
Build failure	CI pipeline	Auto-abort and notify	Any, Team Lead
Test failure	Test suite	Quarantine, fix-forward	After 3 attempts, Architecture
API timeout	Network or 429	Exponential backoff 3x	All retries fail, Manual queue
Policy violation	Validation gate	Reject with explanation	Repeated, Process review
Missing dependency	Graph analysis	Batch hold	> 24 h, Developer escalation

S4.2 Auto-Recovery Patterns

Transient failures get retried with exponential backoff. CI timeouts retry with a fresh build cache. Network failures get queued with idempotent delivery.

Partial failures stay isolated. Success in branch A plus failure in branch B leads to two independent states. The framework does not roll back the success in A. Developer discretion decides whether to revert.

S5. Gap Analysis: Finding Missed Propagations

Despite automation, manual cherry-picks and bypassed processes still happen. A developer in a hurry runs a local cherry-pick and pushes without going through the framework. The result is a missed synchronous propagation. The fix lands in main and never reaches `release/X`.

The gap-analysis subsystem runs a retrospective audit on a daily schedule.

1. Take the latest tag on each branch as a baseline reference point.
2. Compute the commit delta between source and target since the baseline.
3. For each delta commit, check ticket metadata to see whether the commit should have propagated.
4. If the ticket indicates a synchronous propagation requirement, create an auto-reconciliation synchronous propagation. If ambiguous, flag for manual review.

The missed-commit rate is reported as a quality metric on the executive dashboard.

S6. Observability and SLA Framework

S6.1 Velocity Metrics

Metric	Definition	Target SLA
Synchronous propagation lead time	Source merge to all targets merged	P95 < 24 hours
Branch latency	Time to reach a specific target	P95 < 4 hours
Queue depth	Open propagations per branch pair	< 10 per release

S6.2 Quality Metrics

Metric	Target
Synchronous propagation success rate	> 90 percent
Conflict rate	< 5 percent
Regression rate	< 1 percent
Missed commit rate	< 0.1 percent

S6.3 Alerting Hierarchy

Alert	Condition	Recipient
P1	Queue depth > 100 changes	Team + Manager
P2	Synchronous propagation > 48 hours old	Developer + Lead

Alert	Condition	Recipient
P3	Conflict rate > 10 percent	Team
P4	SLA breach trending	Manager

S7. Implementation Roadmap

The framework rolls out in five phases of about four weeks each.

- **Phase 1, foundation.** Git hook implementation. Relational schema. Single source to single target synchronous propagation. Email notifications.
- **Phase 2, scale.** Multi-target synchronous propagation. Risk-based routing. Time-series storage and basic dashboards.
- **Phase 3, intelligence.** Predictive conflict detection. Dependency-aware batching. Document store and full audit. Gap analysis. SLA monitoring.
- **Phase 4, engagement.** Intelligent follow-up system. Multi-channel notifications. Behavioural nudges. Personal developer dashboards.
- **Phase 5, optimisation.** Machine learning for risk classification. Automated conflict resolution. Cross-organisational benchmarking. Data warehouse for long-term trend analysis.

S8. Future Directions

- **NLP extraction.** Synchronous propagation intent from free-text commit messages. Anomaly detection over historical patterns. Recommendation engine for optimal timing windows.
- **Ecosystem integration.** IDE plugins showing pre-commit branch impact. ChatOps for status queries. Mobile alerts for critical blockers.
- **Advanced conflict resolution.** AST-level semantic diff. Three-way merge intelligence informed by test outcomes.
- **Cross-organisational standards.** Open data schema for synchronous propagation events. Federated analytics across organisations without sharing source code.

Appendix A. Maturity Model

Level	Characteristics
1. Manual	Ad-hoc cherry-picks, no tracking, high regression risk
2. Reactive	Basic automation + notifications, manual conflict resolution, limited visibility
3. Managed	Policy-driven synchronous propagation, risk classification, SLA monitoring, stakeholder reporting

Level	Characteristics
4. Predictive	Conflict prediction, dependency batching, gap analysis, predictive alerts, data mining pipeline
5. Autonomous	AI risk classification, intelligent follow-up, auto-resolution, self-healing, prescriptive analytics

---

Appendix B. Complete Event Schema

```
propagation_event:
  event_id: UUID
  timestamp: ISO8601
  commit:
    hash: SHA
    message: string
    author: string
    team: string
    files_changed: list of strings
    lines_added: integer
    lines_removed: integer
  ticket:
    id: string
    type: feature | bug | story
    priority: low | medium | high | critical
    fix_versions: list of strings
  source_branch: string
  target_branches: list of strings
  risk_profile: low | medium | high
  status: pending | propagating | merged | conflict | abandoned
  timeline:
    source_merged: timestamp
    target_requested: timestamp
    review_started: timestamp
    target_merged: timestamp
  metrics:
    lead_time_seconds: integer
    conflict_resolution_seconds: integer
    review_rounds: integer
  followup:
    notifications_sent: integer
    escalations_triggered: integer
    time_to_first_action: integer
    notification_channels: list of strings
  reviewers:
    assigned: list of strings
    approved: list of strings
    requested_changes: list of strings
```

Appendix C. Sample Analytical Queries

Query 1. Overall success rate.

```
SELECT ROUND(100.0 * SUM(CASE WHEN status='MERGED' THEN 1 ELSE 0 END)
       / COUNT(*), 2) AS success_rate
FROM propagation_events;
```

On the reference dataset this returns 80.00 percent.

Query 2. Top developers by volume.

```
SELECT dev_id, COUNT(*) AS propagations
FROM propagation_events
GROUP BY dev_id
ORDER BY propagations DESC
LIMIT 5;
```

Query 3. Risk-adjusted success rate.

```
SELECT risk_profile,
       COUNT(*) AS total,
       SUM(CASE WHEN status='MERGED' THEN 1 ELSE 0 END) AS merged,
       ROUND(100.0 * SUM(CASE WHEN status='MERGED' THEN 1 ELSE 0 END)
              / COUNT(*), 2) AS success_rate
FROM propagation_events
GROUP BY risk_profile;
```

Query 4. Branch pair SLA compliance.

```
SELECT source_branch, target_branch,
       ROUND(100.0 * SUM(CASE WHEN lead_time_hrs < 24 THEN 1 ELSE 0 END)
              / COUNT(*), 2) AS sla_compliance_pct
FROM propagation_events
WHERE status='MERGED'
GROUP BY source_branch, target_branch;
```

Appendix D. Follow-Up Message Templates

Critical, Tier 1.

```
[CRITICAL] Propagation blocked
[JIRA-1234] Security fix needs immediate action.
Blocking 2 downstream PRs.
In queue for 3 days.
```

```
Conflict probability HIGH.
Actions: View PR. Merge. Reassign. Snooze 1 hour.
```

Urgent, Tier 2.

```
[URGENT] 3 changes ready
1. [JIRA-9012] Doc update, ready, 2 minute effort.
2. [JIRA-3456] Test coverage, 1 approval pending.
3. [JIRA-7890] Dependency bump, auto-merge available.
Actions: Review all. View dashboard.
```

Standard, Tier 3.

```
[DIGEST] Weekly summary
2 items requiring attention.
1 in review, 2 days old.
1 WIP awaiting your ready signal.
Team velocity, 85th percentile this sprint.
Actions: View dashboard.
```

Appendix E. Reproducibility Statement

The synthetic dataset accompanying the main paper is generated by `generate_dataset.py` with `random.seed(42)`. The script is deterministic on any Python 3.9 or higher interpreter. The CSV file `propagation_events.csv` is the canonical reference. The validation script `validate_dataset.py` re-derives every distribution table from the CSV. The four dashboards in the main paper figures are produced by `build_dashboards.py` from the same CSV.

Companion files distributed with the main paper:

- `propagation_events.csv`, 500-row dataset
- `generate_dataset.py`, deterministic generator
- `validate_dataset.py`, distribution validator
- `build_dashboards.py`, dashboard image generator
- `dashboard_1_overall.png` through `dashboard_4_repo.png`, the four dashboard images

Online Repository and Visual Companion

The complete artifact set is hosted at:

> **Repository:** https://github.com/brettleehari/SyncForwardProp

The repository layout is:

```
SyncForwardProp/
  paper/        Both PDFs
  data/         propagation_events.csv
  code/         generate_dataset.py, validate_dataset.py, build_dashboards.py
  dashboards/   The four PNG dashboards
  docs/         GitHub Pages site source
```

A browser-based visual companion is served by GitHub Pages from `docs/` at:

> **Visual companion:** https://brettleehari.github.io/SyncForwardProp/

The visual page presents the four dashboards, the workload equation, the production results, and direct download links for the paper PDFs, the CSV, and the source code. A reader on a phone or laptop sees the same figures without opening the PDF.

Quickstart Reproduction

```
git clone https://github.com/brettleehari/SyncForwardProp.git
cd SyncForwardProp/code
pip install matplotlib
python3 generate_dataset.py
python3 validate_dataset.py
python3 build_dashboards.py
```

The validator prints the distribution table matching the values reported in the main paper. The dashboard renderer writes the four PNG files to `../dashboards/` and they match the figures embedded in the main paper.

Licensing

The paper text, the dataset, and the rendered dashboards are released under CC BY 4.0. The code in the repository is released under the MIT License. See the repository `LICENSE` file for the full text.

End of supplementary material.