Teorema: *Teori dan Riset Matematika*, *11*(2), 243–256, September 2026
p-ISSN 2541-0660, e-ISSN 2597-7237 © 2026
https://jurnal.unigal.ac.id/teorema/article/view/25289

Optimasi Penjadwalan Proses Produksi Menggunakan Algoritma *Enhanced Laxity* untuk SmartEdu *Station*

Noer Fajrin[1], Yuliadi Erdani[2], Muhammad Avila Thirafi[3], Suharyadi Pancono[4], Cyndi Odilia Sumantri[5], Natasya Nareswari[6]

[1,2,3,4]*Program Studi D4 Teknologi Rekayasa Otomasi, Jurusan Teknik Otomasi Manufaktur dan Mekatronika, Politeknik Manufaktur Negeri Bandung, Jl. Kanayakan No. 21, Bandung, Jawa Barat 40135, Indonesia*
[5,6]*Program Studi Teknologi Rekayasa Desain Manufaktur, Jurusan Teknik Perancangan Manufaktur, Politeknik Manufaktur Negeri Bandung, Jl. Kanayakan No. 21, Bandung, Jawa Barat 40135, Indonesia*
E-mail: yul_erdani@yahoo.com

ABSTRACT

Scheduling job execution order on production machines is a critical operational decision impacting throughput and deadline fulfillment. Among dynamic scheduling algorithms, the laxity-based approach with Least Laxity First (LLF) is theoretically optimal but suffers from laxity thrashing, excessive context switching when jobs share equal laxity values. This study implements Enhanced Least Laxity First (ELLF) as an LLF extension with an exclusion shield mechanism to eliminate thrashing, adapted from microsecond-tick processor scheduling to second-tick manufacturing time scale. The algorithm is deployed on a low-cost four-layer IoT architecture using Raspberry Pi with InfluxDB and Grafana monitoring stack, applied to a bolt induction-forging machine case study. Testing covers two workload scenarios (underload and overload) with eight periodic jobs over five independent runs each. Results show ELLF achieves zero preemption under overload (100% reduction from LLF's 1.28 preemptions/job), with trade-offs of about 15% waiting time increase and 232 ms average tardiness. The IoT architecture distributes scheduler telemetry to InfluxDB with sub-second latency without disrupting performance. This work empirically demonstrates, through software simulation testing, that the laxity-based approach, particularly ELLF, is effective for soft real-time production scheduling on low-cost IoT architectures. The scheduling system is intended for integration into the SmartEdu Station.

Keywords: *Enhanced Least Laxity First, Internet of Things, laxity-based scheduling, production scheduling, raspberry Pi*

ABSTRAK

Penjadwalan urutan eksekusi *job* pada mesin produksi merupakan keputusan operasional krusial yang mempengaruhi *throughput* dan pemenuhan tenggat waktu. Di antara algoritma penjadwalan dinamis, pendekatan berbasis *laxity* dengan *Least Laxity First* (LLF) secara teoritis optimal namun mengalami *laxity thrashing*, yaitu peralihan konteks berlebihan ketika beberapa *job* memiliki nilai *laxity* yang sama. Penelitian ini mengimplementasikan *Enhanced Least Laxity First* (ELLF) sebagai perluasan LLF dengan mekanisme *exclusion shield* untuk menghilangkan *thrashing*, yang diadaptasi dari penjadwalan prosesor berskala *tick* mikrodetik ke skala waktu manufaktur berbasis *tick* detik. Algoritma diterapkan pada arsitektur IoT empat lapis berbiaya rendah menggunakan *Raspberry Pi* dengan tumpukan pemantauan InfluxDB dan Grafana, pada studi kasus mesin *induction-forging baut*. Pengujian mencakup dua skenario beban (*underload* dan *overload*) dengan delapan *job* periodik selama lima kali pengulangan independen. Hasil menunjukkan ELLF mencapai nol *preemption* pada kondisi *overload* (penurunan 100% dari 1,28 *preemption/job* pada LLF), dengan *trade-off* berupa kenaikan waktu tunggu sekitar 15% dan tardiness rata-rata 232 ms. Arsitektur IoT mendistribusikan telemetri penjadwal ke InfluxDB dengan latensi sub-detik tanpa mengganggu kinerja. Melalui pengujian simulasi perangkat lunak, penelitian ini membuktikan secara empiris bahwa pendekatan berbasis *laxity*, khususnya ELLF, efektif untuk penjadwalan produksi *soft real-time* pada arsitektur IoT berbiaya rendah. Sistem penjadwalan ini ditujukan untuk diintegrasikan pada SmartEdu Station.

Kata kunci: *Enhanced Least Laxity First , Internet of Things,* penjadwalan berbasis *laxity,* penjadwalan produksi, *Raspberry Pi*

Dikirim: 7 Juni 2026; Diterima: 3 Juli 2026; Dipublikasikan: 20 September 2026
Cara sitasi: Fajrin, N., Erdani, Y., Thirafi, M. A., Pancono, S., Sumantri, C. O., & Nareswari, N. (2026). Optimasi Penjadwalan Proses Produksi Menggunakan Algoritma *Enhanced Laxity* untuk SmartEdu. *Station Teorema: Teori dan Riset Matematika, 11(2), 243–256. DOI: http://dx.doi.org/10.25157/teorema.v11i2.25289.*

PENDAHULUAN

Penjadwalan urutan eksekusi *job* pada mesin produksi merupakan keputusan operasional kritis yang berdampak langsung pada *throughput*, pemenuhan tenggat pengiriman, dan efisiensi pemanfaatan aset manufaktur. Pada lantai produksi modern dengan campuran *job* bervariasi tinggi dan bervolume rendah, urutan optimal harus dihitung ulang setiap kali komposisi antrean berubah, sehingga pendekatan penjadwalan statis berbasis aturan seperti *Rate Monotonic Scheduling* (RMS) menjadi tidak fleksibel terhadap fluktuasi beban kerja (Zulfadlillah et al., 2024). Penjadwalan dinamis, yang menghitung prioritas *job* saat *runtime*, hadir sebagai alternatif yang lebih adaptif, di mana keputusan eksekusi dibuat berdasarkan parameter temporal yang berubah terhadap waktu (Kocsi et al., 2020; Rahmah et al., 2024; Zhang, 2025; Zhao et al., 2022).

Salah satu kelas algoritma penjadwalan dinamis yang banyak dikaji adalah pendekatan berbasis *laxity*, yang memprioritaskan *job* berdasarkan nilai *laxity*-nya, didefinisikan sebagai sisa waktu hingga tenggat dikurangi sisa waktu eksekusi. Algoritma paling klasik dalam kategori ini adalah *Least Laxity First* (LLF), yang menjadwalkan *job* dengan *laxity* terkecil terlebih dahulu dan secara teoritis terbukti optimal: setiap himpunan tugas yang dapat dijadwalkan dapat dijadwalkan oleh LLF. Namun, LLF mengalami *laxity thrashing*, yaitu peralihan konteks berlebihan ketika dua *job* atau lebih memiliki nilai *laxity* yang sama (Hildebrandt et al., 1999). Dalam konteks penjadwalan mesin fisik, setiap peralihan konteks berarti menghentikan operasi mesin dan melakukan penyiapan ulang aktuator, yang mahal (hitungan detik hingga menit), sehingga *thrashing* sangat merugikan secara operasional.

Untuk mengatasi keterbatasan ini, Hildebrandt, Golatowski, dan Timmermann (1999) mengusulkan algoritma *Enhanced Least Laxity First* (ELLF), perluasan LLF dalam keluarga algoritma berbasis *laxity* dengan mekanisme tambahan *exclusion shield* untuk menghilangkan *thrashing* (Hildebrandt et al., 1999). Algoritma ini awalnya diperkenalkan untuk penjadwalan prosesor dengan resolusi *tick* mikrodetik. Penelitian ini mengadaptasi algoritma ELLF ke implementasi *user-space* pada platform Raspberry Pi dengan resolusi *tick* 1 detik (sesuai skala waktu manufaktur) dan menerapkannya pada studi kasus penjadwalan mesin *induction-forging baut*. Meskipun ELLF pertama kali diperkenalkan pada tahun 1999, pendekatan berbasis *laxity* tetap menjadi subjek penelitian yang aktif hingga kini, dengan berbagai varian dan analisis mutakhir yang terus diusulkan (Forstmayr & Brunauer, 2021; Han et al., 2023; Kermia, 2022; Younis et al., 2025).

Arsitektur IoT untuk manufaktur umumnya mengikuti model empat lapis: lapisan *sensing*, lapisan jaringan, lapisan *middleware*, dan lapisan aplikasi (Calderón et al., 2024; Farooq et al., 2023; Fatima et al., 2022). Pada arsitektur berbiaya rendah, *Single-Board Computer* seperti Raspberry Pi sering diposisikan sebagai *edge server* yang menjembatani lapisan *sensing* dengan *cloud*, melakukan pra pemrosesan dan agregasi data sebelum transmisi (Daher et al., 2021). Calderón et al. (2024) menunjukkan kelayakan penerapan arsitektur IoT industri menggunakan komponen sumber terbuka yang memadukan perangkat keras dan lunak otomasi serta IoT. Komunikasi antar lapisan dalam konteks IoT industri umumnya ditangani protokol perpesanan ringan (Saha et al., 2024). Untuk mendukung observabilitas dan visualisasi kinerja penjadwal, sistem diintegrasikan dengan tumpukan pemantauan IoT berbiaya rendah berbasis InfluxDB sebagai basis data deret waktu dan Grafana sebagai dasbor visualisasi, tumpukan yang banyak digunakan dalam riset pemantauan IoT (Wang et al., 2023; Wijayaningrum & Wakhidah, 2023).

Beberapa varian LLF telah diusulkan dalam literatur (Forstmayr & Brunauer, 2021; Kermia, 2022), dan algoritma LLF juga telah dieksplorasi sebagai basis penjadwalan komunikasi IoT *real-time* (Deniziak et al., 2023). Pendekatan penjadwalan alternatif seperti *Least Slack Time* (LST) dan studi komparatif algoritma penjadwalan CPU lainnya (Han et al., 2023; Manasa & Mabbu, 2023; Omar et al., 2021) semakin melengkapi konteks riset penjadwalan dinamis yang lebih luas. Studi analisis berbasis data sintetis telah digunakan untuk mengkarakterisasi algoritma penjadwalan *hard* dan *soft real-time* dalam skenario terkendali (Younis et al., 2025).

Meskipun algoritma berbasis *laxity* telah mapan secara teoritis, sebagian besar kajian tersebut masih terbatas pada ranah penjadwalan prosesor dengan biaya peralihan konteks berskala mikrodetik, sehingga *trade-off* dari *laxity thrashing* kerap dianggap tidak signifikan. Masih sedikit penelitian yang mengkaji perilaku ELLF ketika biaya peralihan konteks meningkat drastis, misalnya pada mesin produksi fisik yang penyiapan ulangnya berskala detik hingga menit, maupun ketika algoritma dipindahkan dari koprosesor perangkat keras ber-*tick* mikrodetik ke implementasi *user-space* ber-*tick* detik. Kesenjangan inilah yang menjadi permasalahan utama penelitian ini, yaitu bagaimana mengadaptasi ELLF ke skala waktu manufaktur dan mengukur secara kuantitatif apakah penghapusan *thrashing* tetap menguntungkan pada domain tersebut. Kebaruan penelitian ini terletak pada tiga hal: (1) adaptasi formal definisi keseriaan *laxity* dari kesamaan bit-per-bit menjadi toleransi $|\Delta laxity| \leq 1$ *tick* agar algoritma tahan terhadap *jitter* perangkat lunak; (2) evaluasi kuantitatif *trade-off* ELLF terhadap LLF pada dua kondisi beban dengan biaya peralihan konteks yang relevan secara operasional; dan (3) penyediaan implementasi rujukan pada arsitektur IoT empat lapis berbiaya rendah yang dapat direproduksi. Dengan demikian, kontribusi penelitian ini mencakup aspek teoritis, berupa analisis kompleksitas dan formalisasi keseriaan *laxity*, sekaligus aspek terapan, berupa implementasi dan validasi empiris pada perangkat nyata.

Penelitian ini merupakan bagian dari pengembangan SmartEdu Station, yaitu *plant* edukasi di laboratorium Politeknik Manufaktur Negeri Bandung yang dirancang sebagai sarana pembelajaran otomasi bagi mahasiswa, dengan fokus pada proses *sorting*. Karena SmartEdu Station masih berada dalam tahap pengembangan, stasiun tersebut belum dapat dijadikan lingkungan pengujian yang stabil untuk mengevaluasi algoritma penjadwalan. Oleh karena itu, pengembangan dan validasi algoritma ELLF pada penelitian ini menggunakan proses *induction-forging baut* sebagai studi kasus, karena proses tersebut telah tersedia dan bersifat representatif sebagai proses produksi periodik yang menuntut penjadwalan *real-time*. Algoritma penjadwalan yang dikembangkan bersifat umum terhadap jenis proses karena bekerja atas parameter tugas (waktu eksekusi, tenggat, dan periode), bukan atas jenis prosesnya, sehingga hasil validasi pada studi kasus ini ditujukan untuk diintegrasikan pada SmartEdu Station ketika stasiun tersebut telah beroperasi, sebagai lapisan penjadwalan bagi platform pembelajaran otomasi tersebut.

Tujuan penelitian ini adalah: (1) memvalidasi secara empiris efektivitas pendekatan penjadwalan berbasis *laxity*, khususnya algoritma ELLF, untuk optimasi penjadwalan produksi pada beban kerja periodik, dengan perbandingan kuantitatif terhadap LLF konvensional pada dua kondisi (*underload* dan *overload*); (2) mengkarakterisasi *trade-off* antara pengurangan peralihan konteks dan kenaikan waktu tunggu serta *tardiness* sebagai panduan praktis pemilihan algoritma di domain manufaktur; dan (3) menyediakan implementasi rujukan arsitektur IoT empat lapis berbiaya rendah pada *Single-Board Computer* yang strukturnya memungkinkan integrasi langsung dengan *plant* produksi nyata bila tersedia, divalidasi melalui pengujian berlingkup terbatas dalam simulasi perangkat lunak pada platform yang dibangun.

METODE PENELITIAN

Bagian ini menguraikan dasar algoritma ELLF yang diterapkan, arsitektur sistem IoT yang dibangun, serta prosedur dan parameter pengujian. Penelitian dilakukan dengan pendekatan rekayasa terapan, yaitu mengadaptasi, mengimplementasikan, dan menguji algoritma ELLF pada platform IoT berbiaya rendah, lalu membandingkan kinerjanya terhadap LLF konvensional pada dua skenario beban.

Algoritma ELLF

Sebelum membahas implementasi program, beberapa rumus dasar perlu didefinisikan agar pembahasan hasil pengujian dapat dipahami dengan jelas. Setiap *job* τ_i dalam penjadwalan dicirikan oleh tiga parameter: waktu eksekusi C_i, tenggat relatif D_i, dan periode P_i. Untuk *job* dengan tenggat implisit ($D_i = P_i$), utilisasi *job* tersebut didefinisikan sebagai:

$$U_i = C_i / P_i \quad (1)$$

Total utilisasi seluruh himpunan tugas adalah:

$$U = \Sigma (C_i / P_i) \quad (2)$$

Kondisi $U \leq 1$ disebut *underload* (himpunan tugas secara teoritis dapat dijadwalkan); $U > 1$ disebut *overload* (sebagian tenggat pasti terlewati). Pada waktu t, *laxity job* τ_i, yaitu sisa kelonggaran waktu sebelum tenggat terlewati, dihitung sebagai:

$$L_i\ (t) = (d_i\ - t) - c_i\ (t) \quad (3)$$

dengan d_i adalah tenggat absolut dan $c_i(t)$ adalah sisa waktu eksekusi. Algoritma LLF memilih *job* dengan $L_i(t)$ terkecil pada setiap *tick* (Cottet et al., 2002). Ketika dua *job* memiliki *laxity* sama, LLF memicu peralihan konteks berulang (*laxity thrashing*). ELLF mengatasinya dengan memberi status *excluded* pada *job* yang seri: *job* semacam itu hanya boleh menyela jika *laxity*-nya menjadi lebih kecil daripada *laxity job* yang berjalan dan lebih kecil daripada ambang himpunan *excluded*.

LLF, ELLF, dan Exclusion Shield

ELLF menghilangkan peralihan bolak-balik dengan memperkenalkan himpunan *exclusion* E(t). Ketika penjadwalan memilih tugas τ_run dari kelompok yang memiliki *laxity* terkecil, semua tugas lain dalam kelompok tersebut ditambahkan ke E(t); nilai *laxity* mereka tetap diperbarui pada setiap *tick*, tetapi mereka dilarang menyela τ_run. Aturan *preemption* ELLF adalah:

$$preempt(\tau\_run) \Leftrightarrow \exists\ \tau \in (\Gamma - E(t)) : L\_\tau(t) < L\_\tau run(t) \wedge L\_\tau(t) < min\ L\_\sigma(t) \quad (4)$$

Persamaan (4) secara formal menyatakan bahwa *preemption* hanya diizinkan jika terdapat tugas non-*excluded* yang *laxity*-nya lebih rendah daripada *laxity* tugas yang berjalan sekaligus lebih rendah daripada *laxity* terendah pada himpunan *excluded*. Dengan kata lain, sebuah kandidat harus memenuhi dua ambang: *laxity* tugas yang berjalan, dan *laxity* terendah di antara tugas yang dilindungi. Sebagai contoh, jika tugas yang berjalan memiliki *laxity* 5 dan dua tugas yang dilindungi memiliki *laxity* 6 dan 8, maka kandidat harus memiliki *laxity* lebih kecil dari 5 untuk dapat memicu *preemption*. Tugas *excluded* memperoleh kembali kelayakannya hanya ketika *τ_run* selesai atau ketika *τ_run* sendiri menjadi *excluded* berdasarkan (4). Hildebrandt et al. (1999) menunjukkan bahwa dengan mekanisme ini, jumlah peralihan konteks pada LLF, yang tumbuh *non-linier* terhadap jumlah *job* yang seri, turun menjadi:

$$N\_cs(ELLF) = n \quad (5)$$

untuk sekelompok n *job* yang seri. Logika inti pemilihan, yang secara langsung mengimplementasikan aturan *preemption* ELLF, ditunjukkan pada Gambar 1.

```
// (a) Threshold dari excluded tasks
long long min_slack_excluded = LLONG_MAX;
for (int i = 0; i < num_tasks_loaded; i++) {
    if (!tasks[i].is_complete && tasks[i].is_active && tasks[i].is_excluded) {
        if (tasks[i].current_laxity_us < min_slack_excluded) {
            min_slack_excluded = tasks[i].current_laxity_us;
        }
    }
}

// (b) Kandidat terbaik dari non-excluded tasks
long long min_slack_valid = LLONG_MAX;
long long earliest_dl_valid = LLONG_MAX;
int best_valid_task = -1;

for (int i = 0; i < num_tasks_loaded; i++) {
    if (!tasks[i].is_complete && tasks[i].is_active &&
        !tasks[i].is_excluded && now_us >= tasks[i].release_time_us) {

        if (tasks[i].current_laxity_us < min_slack_valid) {
            min_slack_valid = tasks[i].current_laxity_us;
            earliest_dl_valid = tasks[i].absolute_deadline_us;
            best_valid_task = i;
        } else if (llabs(tasks[i].current_laxity_us - min_slack_valid) <= LAXITY_TIE_WINDOW_US) {
            if (tasks[i].absolute_deadline_us < earliest_dl_valid) {
                earliest_dl_valid = tasks[i].absolute_deadline_us;
                best_valid_task = i;
            }
        }
    }
}

// (c) Keputusan scheduling
int selected = current_running_task;

if (best_valid_task == -1) {
    // Tidak ada kandidat valid — idle (atau tetap di running task kalau ada)
} else if (current_running_task == -1) {
    selected = best_valid_task;
} else if (best_valid_task == current_running_task) {
    selected = current_running_task;
} else {
    // [ELLF Rule 3] Preempt HANYA jika kandidat lebih genting dari
    // KEDUA running task MAUPUN threshold excluded.
    long long running_slack = tasks[current_running_task].current_laxity_us;
    if (min_slack_valid < running_slack && min_slack_valid < min_slack_excluded) {
        selected = best_valid_task;
    } else {
        selected = current_running_task;
    }
}
```

Gambar 1. Logika inti pemilihan ELLF dengan pemindaian dua tahap dan kondisi *preemption exclusion-shield.*

Exclusion shield dijelaskan sebagai satu *flag boolean* is_*excluded* pada struktur data tiap tugas. *Flag* ini diaktifkan setiap kali sebuah tugas seri dengan tugas terpilih dan dinonaktifkan ketika tugas yang berjalan selesai atau ketika pemilihan baru mengevaluasi himpunan *excluded* E(t) yang memenuhi syarat. Mekanisme ini merealisasikan empat aturan ELLF dari Hildebrandt et al. (1999) secara langsung. Keseluruhan program berukuran sekitar 580 baris kode C.

Analisis Kompleksitas Algoritma

Kompleksitas ELLF ditinjau dari bagaimana beban kerja penjadwal berubah ketika jumlah *job* bertambah. Pada setiap *tick*, penjadwal menelusuri himpunan *job* aktif secara berurutan untuk menentukan ambang *laxity* pada himpunan *excluded*, memilih kandidat dengan *laxity* terkecil dari himpunan non-*excluded*, dan mengevaluasi syarat *preemption* pada Persamaan (4). Ketiga langkah tersebut merupakan penelusuran daftar *job* satu per satu, bukan perbandingan setiap *job* terhadap seluruh *job* lainnya. Karena itu, beban komputasi per *tick* bertambah sebanding dengan banyaknya *job*, bukan secara berlipat, sehingga penjadwal tetap praktis meskipun ukuran himpunan *job* diperbesar.

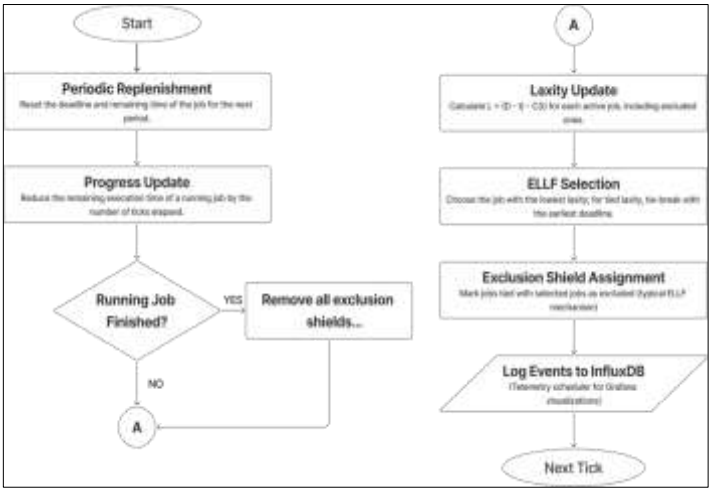
Aspek teoritis yang paling menentukan adalah jumlah peralihan konteks. Pada LLF konvensional, sekelompok *job* dengan *laxity* yang sama saling menyela secara bergantian pada *tick-tick* berikutnya (*laxity thrashing*), sehingga peralihan konteks yang terjadi jauh lebih banyak daripada jumlah *job* yang terlibat. Mekanisme *exclusion shield* memutus siklus ini: setelah satu *job* terpilih, *job-job* lain yang seri untuk sementara dikeluarkan dari pencalonan melalui syarat pada Persamaan (4) sehingga tidak dapat menyela. Untuk sekelompok n *job* yang seri, mekanisme ini menekan jumlah peralihan konteks menjadi tepat sebanyak n, sebagaimana dinyatakan pada Persamaan (5). Dengan demikian, biaya peralihan konteks ELLF terkendali dan sebanding dengan jumlah *job* yang seri, berbeda dengan LLF yang biayanya tidak terkendali pada kondisi serupa.

Kedua sifat di atas, yaitu beban per *tick* yang sebanding dengan jumlah *job* dan jumlah peralihan konteks yang terkendali sesuai Persamaan (5), menjadi dasar teoritis skalabilitas ELLF. Argumen ini melengkapi pengujian empiris yang dilakukan pada delapan *job*: keunggulan utama ELLF, yaitu penekanan peralihan konteks, secara teoritis tetap terjaga seiring bertambahnya ukuran

himpunan *job*, tanpa memerlukan asumsi tambahan di luar mekanisme yang telah diformalkan pada Persamaan (4) dan (5).

Lima Fase Loop Penjadwalan

Algoritma ELLF yang berjalan pada lapisan *middleware* terdiri atas lima fase yang dieksekusi setiap *tick* 1 detik: (1) mengisi ulang tugas periodik yang telah selesai; (2) mengurangi sisa waktu komputasi tugas yang berjalan; (3) menghitung ulang *laxity* untuk semua tugas aktif; (4) memilih tugas berikutnya untuk dijalankan; (5) menerapkan *exclusion shield* pada tugas yang seri dengan tugas terpilih, sebagaimana diilustrasikan pada Gambar 2.

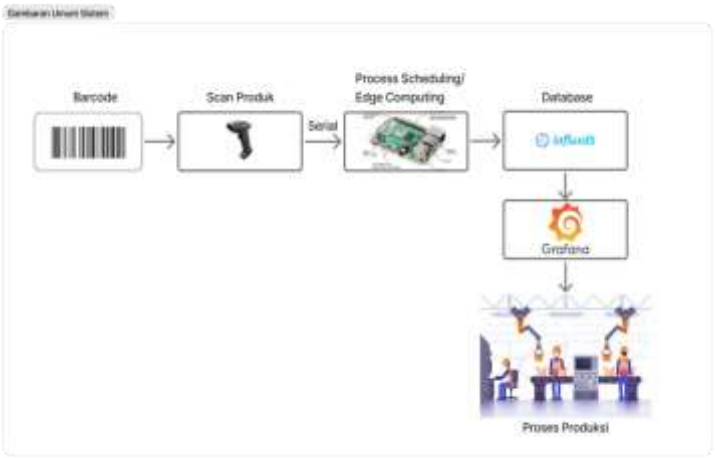


Gambar 2. Diagram alir loop penjadwal ELLF lima *fase* (*tick* = 1 detik).

Adaptasi penting dari makalah Hildebrandt et al. (1999) terletak pada definisi “seri”. Pada implementasi koprosesor perangkat keras aslinya, dua *job* dianggap seri jika nilai *laxity*-nya identik bit-per-bit. Pada implementasi perangkat lunak *user-space* ini, *jitter* dari penjadwalan sistem operasi menyebabkan *laxity job* nyaris tidak pernah persis sama. Oleh karena itu, definisi seri diperluas menjadi $|\Delta laxity| \leq 1$ *tick* (= 1 detik), yang mempertahankan makna makalah sekaligus mengakomodasi karakteristik perangkat lunak *real-time*.

Arsitektur Sistem IoT

Arsitektur sistem dirancang sebagai integrasi *end-to-end* dari titik akuisisi data di lantai produksi hingga dasbor visualisasi, mengikuti pola arsitektur IoT empat lapis. Integrasi antarlapisan dirancang sebagai *pipeline* satu arah dengan lapisan persistensi (InfluxDB) di pusat sebagai *single source of truth*. Diagram alir data ditunjukkan pada Gambar 3.



Gambar 3. Alur integrasi antarlapisan pada arsitektur IoT empat lapis.

Lapisan *sensing* diwakili oleh *barcode scanner* sebagai perangkat lapangan pada level 0 menurut *Purdue Model*. Pemindai membaca parameter instruksi kerja, yaitu waktu eksekusi (C), tenggat (D), periode (P), dan jumlah siklus produksi (Loop), dari kode 2D pada produk, menggunakan format teks "C D P" atau "C D P Loop". Setiap pemindaian menandai kedatangan beban kerja baru. Sensor CMOS yang digunakan beroperasi pada kecepatan pemindaian tinggi dengan akurasi pembacaan hingga 99,9% (Lin & Fuh, 2013; Xie & Tan, 2021).

Pada lapisan jaringan, pemindai bertindak sebagai pengirim serial yang mengirim deret data segera setelah pemindaian berhasil. *Edge server* membaca port serial secara kontinu dalam *receiver loop* dan mengurai setiap baris masuk untuk mengekstrak parameter C, D, P, dan Loop. Port serial yang digunakan adalah /dev/ttyACM0 pada sistem operasi Linux di *edge server*.

Lapisan *middleware* berjalan pada Raspberry Pi, yang memiliki peran ganda sebagai (1) *edge gateway* yang menjembatani lapisan *sensing* dengan lapisan aplikasi dan (2) *edge server* yang menjalankan komputasi inti algoritma penjadwalan. Raspberry Pi dipilih sebagai platform *edge* karena memadukan harga terjangkau, kapabilitas komputasi memadai, dan ekosistem perangkat lunak sumber terbuka yang kaya (Daher et al., 2021; Goyal, 2024; Hassan et al., 2022). Sistem operasi yang digunakan adalah Linux dengan modul penjadwal dibangun dalam bahasa C *user-space* untuk menjamin kendali penuh atas pewaktuan eksekusi. *Middleware* terdiri atas tiga modul yang berjalan dalam satu proses *multi-threaded*: *serial parser*, penjadwal ELLF dengan *tick* 1 detik, dan *worker thread* per tugas. Sinkronisasi antar-*thread* menggunakan *POSIX mutex* dan *condition variable*.

Pada persistensi data, *edge server* mengirim data ke InfluxDB melalui protokol HTTP menggunakan format *line-protocol* InfluxDB. Skema penyimpanan menggunakan dua *measurement* utama: *scheduler_events* untuk merekam interval eksekusi tiap *job*, dan *final performance* untuk merekam metrik *agregat per job* pada akhir *batch*. Transmisi dilakukan secara asinkron dengan *timeout* singkat (50 ms untuk *event log*, 1000 ms untuk laporan akhir) agar tidak mengganggu kinerja penjadwal. InfluxDB dipilih karena dioptimalkan untuk skenario *write-intensive* dan *query-intensive* pada *dataset* yang tumbuh dengan laju tinggi (Wang et al., 2023).

Pada lapisan aplikasi, *dashboard Grafana* menampilkan tiga panel utama: (1) *Gantt chart* yang memvisualisasikan urutan eksekusi *job* pada lini waktu *real-time*, (2) tabel kinerja yang menampilkan metrik akhir tiap *job*, dan (3) panel peringatan yang memberi peringatan visual ketika sistem mendekati kondisi *overload*. Lapisan aplikasi membaca data langsung dari InfluxDB menggunakan *data polling* via kueri HTTP berbasis bahasa Flux (Wijayaningrum & Wakhidah, 2023).

Pengaturan Pengujian

Pengujian dirancang untuk memvalidasi (1) fungsionalitas integrasi antar lapisan IoT dan (2) efektivitas algoritma ELLF dibandingkan LLF konvensional. Konfigurasi pengujian dirangkum pada Tabel 1.

Tabel 1. Konfigurasi pengujian kinerja

Parameter	Nilai
Algoritma yang diuji	LLF konvensional dan ELLF (Hildebrandt et al., 1999)
Jumlah job	8 job periodik (Task_1 – Task_8)
Waktu eksekusi per job	5 detik (skala uji); produksi nyata: 5 menit
Skenario beban	Underload (U ≤ 100%) dan *Overload* (U > 100%, hingga ~122%)
Tick penjadwal	1 detik
Pengulangan per skenario	5 kali independen

Keenam parameter pada Tabel 1 dipilih sebagai berikut. Algoritma yang diuji dibatasi pada LLF konvensional dan ELLF karena fokus penelitian adalah mengisolasi pengaruh *exclusion shield*; LLF menjadi algoritma dasar sekaligus algoritma induk yang diperluas ELLF, sehingga selisih kinerja

keduanya dapat diatribusikan langsung pada mekanisme tersebut. Jumlah *job* ditetapkan delapan agar cukup besar untuk memunculkan keseriaan *laxity*, pemicu *thrashing*, namun tetap jelas divisualisasikan pada *Gantt chart*. Waktu eksekusi per *job* 5 detik dipakai sebagai skala uji yang mewakili produksi nyata 5 menit, dipilih agar durasi pengujian praktis sekaligus mempertahankan rasio terhadap *tick*. Skenario beban dibedakan *underload* dan *overload* untuk mengamati perilaku kedua algoritma di kedua sisi titik kritis utilisasi U = 1. *Tick* penjadwal 1 detik ditetapkan sebagai pendamping proporsional terhadap waktu eksekusi (rasio C/*tick* = 5:1), meniru granularitas keputusan penjadwalan skala manufaktur. Pengulangan lima kali independen per skenario digunakan agar rata-rata metrik stabil terhadap variasi akibat *jitter* penjadwalan sistem operasi.

Rincian *dataset* masukan untuk kedua skenario pengujian (*underload* dan *overload*) disajikan pada Tabel 2 dan Tabel 3. Setiap baris mewakili satu *job* periodik dengan parameter waktu eksekusi, periode, dan utilisasi individual. *Tick* penjadwal ditetapkan 1 detik sebagai pendamping proporsional terhadap waktu eksekusi 5 detik yang digunakan pada eksperimen ini. Rasio C/*tick* sebesar 5:1 dipertahankan baik pada skala uji (C = 5 detik, *tick* = 1 detik) maupun pada skala produksi nyata (C = 5 menit, *tick* = 1 menit), sehingga hasil dapat diekstrapolasi secara linier dengan faktor 60 untuk metrik berbasis waktu; metrik berbasis hitungan tetap (Younis et al., 2025).

Tabel 2. Dataset masukan skenario underload (utilisasi total U = 0,7705)

Task	C_i (s)	$P_i = D_i$ (s)	U_i
τ_1	5	40	0,125
τ_2	5	45	0,1111
τ_3	5	50	0,1
τ_4	5	50	0,1
τ_5	5	55	0,0909
τ_6	5	60	0,0833
τ_7	5	60	0,0833
τ_8	5	65	0,0769
Total U			0,7705

Kedua skenario dirancang untuk mengapit titik kritis utilisasi U = 1. Skenario *underload* pada Tabel 2 menetapkan utilisasi total U = 0,7705 yang mewakili beban kerja normal dan secara teoretis dapat dijadwalkan sepenuhnya, sedangkan skenario *overload* pada Tabel 3 menaikkan utilisasi hingga U = 1,221 dengan memperpendek periode setiap *job* sehingga sebagian tenggat dipastikan terlewati dan perilaku kedua algoritma pada kondisi jenuh dapat diamati.

Tabel 3. Dataset masukan skenario *overload* (utilisasi total U = 1,221)

Task	C_i (s)	$P_i = D_i$ (s)	U_i
τ_1	5	25	0,200
τ_2	5	28	0,179
τ_3	5	30	0,167
τ_4	5	32	0,156
τ_5	5	35	0,143
τ_6	5	38	0,132
τ_7	5	40	0,125
τ_8	5	42	0,119
Total U			1,221

Parameter pengukuran yang direkam meliputi waktu tunggu (lama *job* menunggu dalam antrean siap, ms), *tardiness* (keterlambatan relatif terhadap tenggat, ms), utilisasi CPU (%), *miss rate* (%), *preemption* (frekuensi peralihan konteks per *job*), dan *throughput* (*job*/s). Semua parameter dihitung dari *log* yang dipersistenkan ke InfluxDB sehingga tidak diperlukan perhitungan manual.

HASIL DAN PEMBAHASAN

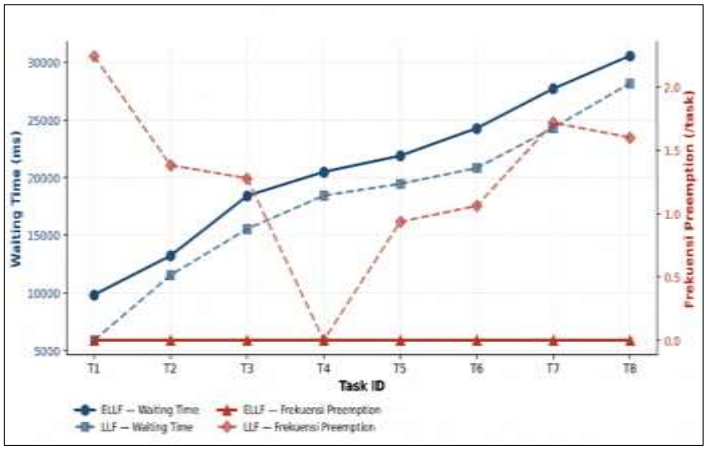
Hasil kinerja terdapat pada rangkuman hasil rata-rata dari 5 pengulangan × 8 tugas untuk kedua skenario disajikan pada Tabel 4.

Tabel 4. Hasil pengujian rata-rata gabungan

Algoritma	Kondisi	Wait (ms)	Tardy (ms)	CPU (%)	Miss (%)	Preempt
ELLF	Underload	15.180	0,00	12,56	0,00	0,00
LLF	Underload	13.267	0,00	12,75	0,00	0,00
ELLF	*Overload*	20.805	232,45	15,98	12,49	0,00
LLF	*Overload*	18.030	79,95	16,27	8,33	1,28

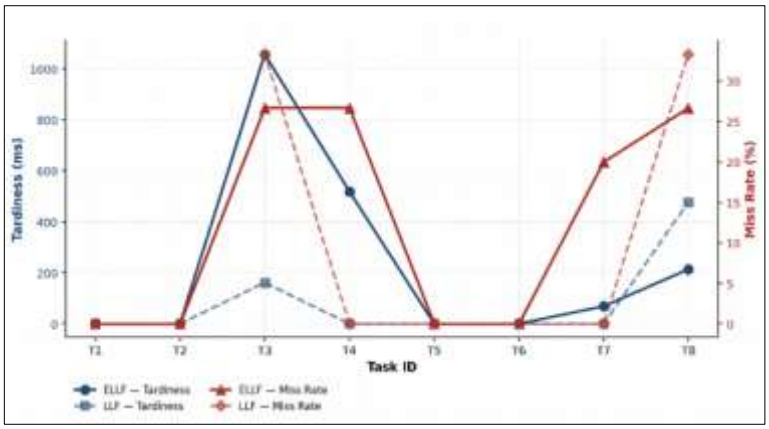
Pada kondisi *underload*, kedua algoritma mempertahankan *miss rate* nol, *tardiness* nol, dan *preemption* nol. Mekanisme *exclusion shield* ELLF tidak menimbulkan *overhead* operasional yang berarti pada wilayah aman. Pada kondisi *overload*, ELLF mencapai *preemption* nol (penurunan 100% dari 1,28 pada LLF), sementara LLF mempertahankan *preemption* agresif yang menghasilkan *tardiness* lebih kecil. *Trade-off*-nya jelas: ELLF mengorbankan sekitar 15% waktu tunggu dan 152 ms *tardiness* (selisih terhadap LLF) sebagai pertukaran untuk menghilangkan peralihan konteks. Gambar 4 menegaskan efektivitas *exclusion shield* ELLF dalam menghilangkan *laxity thrashing*: waktu tunggu kedua algoritma sebanding, tetapi ELLF menekan *preemption* hingga nol pada kondisi *overload*, berbeda dengan LLF yang memicu rata-rata 1,28 *preemption* per tugas.

Pola hasil ini selaras dengan karakterisasi teoretis penjadwalan berbasis *laxity* pada literatur. Kemampuan LLF mempertahankan *miss rate* nol pada kondisi *underload* konsisten dengan sifat optimalitas dinamisnya yang dilaporkan pada studi komparatif algoritma penjadwalan *real-time* (Manasa & Mabbu, 2023; Omar et al., 2021). Namun, kecenderungan LLF memicu peralihan konteks berlebih pada kondisi jenuh, yang pada penelitian ini terukur sebesar 1,28 *preemption* per *job*, merupakan manifestasi empiris dari *laxity thrashing* yang diprediksi Hildebrandt et al. (1999) dan menjadi motivasi utama pengusulan berbagai varian LLF (Forstmayr & Brunauer, 2021; Kermia, 2022). Temuan bahwa *exclusion shield* menekan *preemption* hingga nol tanpa menurunkan *miss rate* pada *underload* memperkuat argumen bahwa penstabilan peralihan konteks tidak harus mengorbankan optimalitas penjadwalan pada beban normal. Kenaikan waktu tunggu sekitar 15% dan *tardiness* sebesar 152 ms merupakan harga terukur dari penstabilan tersebut; besaran ini sejalan dengan ekspektasi bahwa penundaan *preemption* menggeser sebagian *job* melewati tenggatnya pada kondisi *overload*, sebagaimana juga teramati pada studi penjadwalan berbasis data sintetis (Younis et al., 2025).



Gambar 4. Waktu tunggu dan frekuensi *preemption* pada kondisi *overload*.

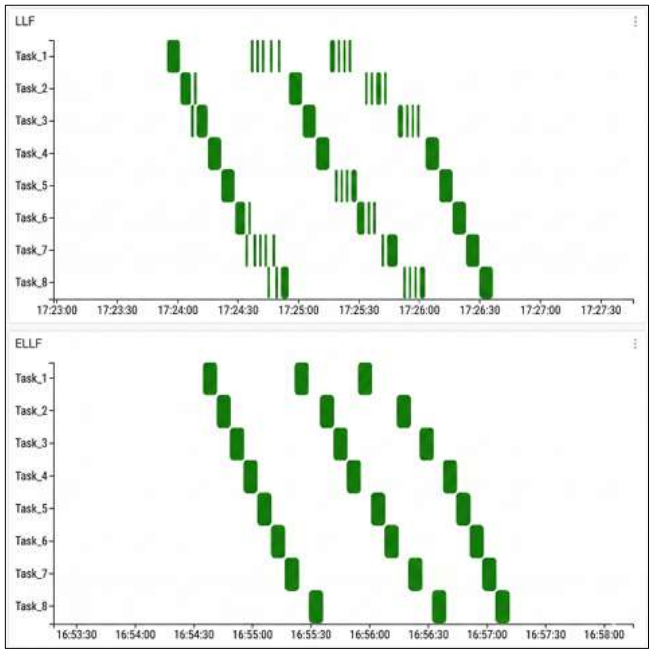
Selanjutnya, Gambar 5 merangkum metrik ketepatan waktu, yaitu *tardiness* dan *deadline miss rate*, untuk kedua algoritma pada kedua kondisi beban. Pada kondisi *underload*, keduanya tidak mengalami keterlambatan maupun kegagalan tenggat. Perbedaan baru tampak pada kondisi *overload*: ELLF mencatat *tardiness* rata-rata 232,45 ms dan *miss rate* 12,49%, sedikit lebih tinggi daripada LLF (79,95 ms dan 8,33%). Selisih ini merupakan konsekuensi langsung dari penundaan *preemption* oleh *exclusion shield* yang menahan sebagian *job* lebih lama ketika sistem jenuh.



Gambar 5. Tardiness dan deadline miss rate pada kondisi *overload*.

Visualisasi Gantt Chart

Perbedaan perilaku antara LLF dan ELLF pada kondisi *overload* paling jelas terlihat melalui visualisasi *Gantt chart* yang ditangkap secara *real-time* dari dasbor Grafana. Gambar 6 menunjukkan perbandingan lini waktu eksekusi LLF (atas) dan ELLF (bawah) pada kondisi *overload* yang sama. Grafana yang membaca data *measurement* scheduler_events langsung dari InfluxDB melalui kueri Flux. Setiap batang horizontal merepresentasikan satu interval eksekusi *job*, dengan sumbu mendatar sebagai lini waktu nyata dan sumbu tegak sebagai daftar *job* (Task_1 hingga Task_8). Karena penjadwal mengirim setiap kejadian eksekusi ke InfluxDB pada saat segmen selesai, dasbor memperbarui tampilan *Gantt chart* secara *near real-time* seiring berjalannya proses penjadwalan. Dengan demikian, panel ini sekaligus berfungsi sebagai sarana pemantauan operasional sekaligus sebagai bukti visual perilaku kedua algoritma, tanpa memerlukan pengolahan data tambahan di luar tumpukan IoT yang dibangun.



Gambar 6. Gantt chart LLF (atas) vs ELLF (bawah) pada kondisi *overload*.

Pada bagian LLF di Gambar 6, terlihat kelompok-kelompok batang vertikal tipis pada Task_1, Task_3, Task_5, Task_7, dan Task_8. Setiap kelompok memuat 4–5 garis vertikal berdampingan yang mewakili fragmen eksekusi yang terputus oleh *preemption*. Pola ini adalah ciri visual khas *laxity thrashing*. Pada bagian ELLF, untuk konfigurasi *job* yang identik, setiap *job* dieksekusi sebagai satu batang utuh tanpa fragmentasi. Mekanisme *exclusion shield* ELLF berhasil mencegah peralihan berulang yang sia-sia: selama sebuah *job* berjalan, *job* lain yang seri dalam *laxity* ditahan hingga *job* yang berjalan selesai. Hasilnya adalah lini waktu eksekusi yang bersih dan kontinu, persis seperti yang diprediksi Hildebrandt et al. (1999). ELLF menghasilkan N_cs = n peralihan konteks (satu per *job*) alih-alih ratusan peralihan seperti LLF. Visualisasi ini berfungsi sebagai validasi empiris atas landasan matematis pada Persamaan (4).

Validasi Arsitektur IoT

Arsitektur IoT yang dibangun terbukti mampu menjalankan algoritma penjadwalan dinamis secara stabil sepanjang pengujian. Latensi transmisi data dari *edge server* ke InfluxDB konsisten pada rentang sub-detik, sehingga keterlambatan jaringan atau kecepatan tulis basis data tidak mengganggu kinerja penjadwal. Dasbor Grafana berhasil memvisualisasikan *Gantt chart* eksekusi *job* secara *real-time*, memungkinkan pengamatan langsung fenomena *thrashing* pada LLF dan penghapusannya pada ELLF. Hal ini menegaskan bahwa kombinasi *Raspberry* Pi + InfluxDB + Grafana merupakan platform yang layak untuk memantau penjadwalan produksi pada skala laboratorium.

Trade-off ELLF vs LLF

Pada implementasi penjadwalan CPU klasik, *trade-off* ELLF (kenaikan 15% waktu tunggu, 152 ms kenaikan *tardiness*) dapat dianggap kurang menguntungkan karena biaya peralihan konteks sangat kecil (orde mikrodetik). Secara teoretis, dalam konteks penjadwalan mesin produksi yang melibatkan aktuator fisik, setiap peralihan konteks berarti menghentikan aktuator fisik (mesin *induction-forging* laboratorium) dan penyiapan ulang yang memakan waktu hitungan detik hingga menit. Walaupun pengujian dalam penelitian ini dilakukan secara simulasi perangkat lunak tanpa aktuator fisik dalam lingkaran kendali, ekstrapolasi hasil ke konteks tersebut menunjukkan bahwa *tardiness* berskala milidetik jauh lebih murah daripada satu kali penyiapan ulang mesin, sehingga ELLF dapat dianggap pilihan yang menguntungkan untuk domain manufaktur.

Keterbatasan Skala Uji

Pengujian dilakukan dengan waktu eksekusi per *job* sebesar 5 detik (skala uji), bukan 5 menit seperti pada produksi *induction-forging baut* yang sebenarnya, demi efisiensi pengumpulan data. Akibatnya, rasio biaya *preemption* terhadap biaya eksekusi selama pengujian lebih besar daripada kondisi produksi nyata. Pada skala 5 menit, keunggulan ELLF dalam menghilangkan *preemption* diprediksi jauh lebih signifikan dari sisi biaya operasional.

Selain keterbatasan skala waktu, pengujian empiris pada penelitian ini terbatas pada himpunan delapan *job*. Argumen teoretis pada subbagian Analisis Kompleksitas Algoritma menunjukkan bahwa beban penjadwal bertambah sebanding dengan jumlah *job* dan jumlah peralihan konteks tetap terkendali sesuai Persamaan (5), sehingga algoritma secara teoretis skalabel terhadap jumlah *job* yang lebih besar. Meskipun demikian, validasi empiris pada himpunan *job* berukuran besar, misalnya puluhan hingga ratusan *job*, belum dilakukan. Pengujian skalabilitas empiris semacam itu, termasuk pengukuran konsistensi performa pada beragam pola beban, menjadi arah pengembangan berikutnya untuk melengkapi jaminan skalabilitas analitis yang telah ditunjukkan.

KESIMPULAN

Penelitian ini telah berhasil mengimplementasikan dan mengevaluasi algoritma *Enhanced Least Laxity First* (ELLF) sebagai pendekatan penjadwalan berbasis *laxity* untuk optimasi proses produksi pada studi kasus mesin *induction-forging baut*. Algoritma diimplementasikan pada arsitektur IoT empat lapis berbiaya rendah berbasis Raspberry Pi dengan tumpukan InfluxDB sebagai basis data deret waktu dan Grafana sebagai dasbor, yang terbukti mampu menyediakan platform pemantauan penjadwalan *real-time* dengan latensi sub-detik. Algoritma ELLF berhasil menghilangkan *laxity thrashing* sepenuhnya pada kondisi *overload* (nol *preemption* dibanding 1,28 pada LLF), dengan *trade-off* berupa kenaikan waktu tunggu sekitar 15% dan *tardiness* 232 ms yang justru menguntungkan untuk penjadwalan mesin fisik. Pada kondisi *underload*, ELLF setara dengan LLF tanpa menambah *overhead*, sehingga *exclusion shield* tidak merugikan di wilayah aman.

REKOMENDASI

Penelitian lanjutan diarahkan pada validasi di *plant* industri nyata dengan aktuator fisik, pengujian pada skala waktu eksekusi 5 menit sesuai kondisi produksi sebenarnya, serta eksplorasi varian *multi-core* dari algoritma ini. Selain itu, integrasi langsung dengan kendali aktuator dan sistem keselamatan *plant* dapat menjadi tahap berikutnya untuk membawa hasil proof-of-concept ini menuju penerapan produksi penuh.

UCAPAN TERIMAKASIH

Penelitian ini didukung oleh Kementerian Pendidikan Tinggi, Sains, dan Teknologi Republik Indonesia melalui Skema Hibah Penelitian Fundamental nomor 136/SPK/C.C4/PPK.DHK/IV/2026. Penulis mengucapkan terima kasih kepada Politeknik Manufaktur Negeri Bandung atas penyediaan sarana dan prasarana laboratorium.

PERNYATAAN PENGGUNAAN KECERDASAN BUATAN (AI)

Dalam penelitian ini, penulis menggunakan Claude AI (Anthropic) secara terbatas, yaitu untuk membantu parafrase kalimat dan penyempurnaan aspek kebahasaan/penulisan. Claude AI tidak digunakan dalam proses analisis data, pengolahan data, pemodelan, maupun pembuatan visualisasi penelitian. Seluruh keluaran AI telah diperiksa, diverifikasi, dan ditinjau oleh penulis. Penulis tetap bertanggung jawab penuh atas keakuratan, orisinalitas, integritas, dan seluruh isi penelitian ini.

DAFTAR PUSTAKA

Calderón, D., Folgado, F. J., González, I., & Calderón, A. J. (2024). Implementation and experimental application of industrial IoT architecture using automation and IoT hardware/software. *Sensors, 24*(24), 8074. https://doi.org/10.3390/s24248074

Cottet, F., Delacroix, J., Kaiser, C., & Mammeri, Z. (2002). *Scheduling in real-time systems*. Wiley.

Daher, A. W., Rizik, A., Muselli, M., Chible, H., & Caviglia, D. D. (2021). Porting Rulex software to the Raspberry Pi for machine learning applications on the edge. *Sensors, 21*(19), 6526. https://doi.org/10.3390/s21196526

Deniziak, S., Płaza, M., & Arcab, Ł. (2023). Real-time communication model for IoT systems. *Annals of Computer Science and Information Systems, 35*, 931–936. https://doi.org/10.15439/2023F8513

Farooq, M. S., Abdullah, M., Riaz, S., Alvi, A., Rustam, F., López Flores, M. A., Castanedo Galán, J., Samad, M. A., & Ashraf, I. (2023). A survey on the role of industrial IoT in manufacturing for implementation of smart industry. *Sensors, 23*(21), 8958. https://doi.org/10.3390/s23218958

Fatima, Z., Tanveer, M. H., Waseemullah, Zardari, S., Naz, L. F., Khadim, H., Ahmed, N., & Tahir, M. (2022). Production plant and warehouse automation with IoT and Industry 5.0. *Applied Sciences, 12*(4), 2053. https://doi.org/10.3390/app12042053

Forstmayr, F., & Brunauer, F. (2021). *Lazy least laxity scheduling for Linux-based programmable logic controller* [Preprint]. Salzburg University of Applied Sciences. https://doi.org/10.13140/RG.2.2.28017.61281

Goyal, Y. (2024). Comparative study of microcontroller: Arduino Uno, Raspberry Pi 4, ESP 32. *International Journal for Research in Applied Science and Engineering Technology, 12*(7), 588–592. https://doi.org/10.22214/ijraset.2024.63598

Han, S., Paik, W., Ko, M.-C., & Park, M. (2023). A comparative study on the schedulability of the EDZL scheduling algorithm on multiprocessors. *Applied Sciences, 13*(18), 10131. https://doi.org/10.3390/app131810131

Hassan, A., Nahar, H., Md Shah, W., Abd-Aziz, A., Sahiran, S. A., Bahaman, N., Ahmad, M. R., Hamid, I. R. A., & Sidik, M. A. B. (2022). Performance evaluation of Raspberry Pi as an IoT edge signal processing device for a real-time flash flood forecasting system. *International Journal of Advanced Computer Science and Applications, 13*(10).

Hildebrandt, J., Golatowski, F., & Timmermann, D. (1999). Scheduling coprocessor for enhanced least-*laxity*-first scheduling in hard real-time systems. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems* (pp. 208–215). https://doi.org/10.1109/EMRTS.1999.777467

Kermia, O. (2022). Strictly periodic first: An optimal variant of LLF for scheduling tasks in a time-critical cyber-physical system. *Concurrency and Computation: Practice and Experience, 34*, e5908. https://doi.org/10.1002/cpe.5908

Kocsi, B., Matonya, M. M., Pusztai, L. P., & Budai, I. (2020). Real-time decision-support system for high-mix low-volume production scheduling in Industry 4.0. *Processes, 8*(8), 912. https://doi.org/10.3390/pr8080912

Lin, J. A., & Fuh, C. S. (2013). 2D barcode image decoding. *Mathematical Problems in Engineering, 2013*, 848276. https://doi.org/10.1155/2013/848276

Manasa, R., & Mabbu, D. (2023). A comparative approach on scheduling algorithms for real-time systems. *Electronics and Computer Express, 1*(1), 29–35. https://doi.org/10.59535/ece.v1i1.11

Omar, H. K., Jihad, K. H., & Hussein, S. F. (2021). Comparative analysis of the essential CPU scheduling algorithms. *Bulletin of Electrical Engineering and Informatics, 10*(5), 2742–2750.

Rahmah, G. M., Fauziah, K., & Suroso, F. (2024). Implementasi metode Earliest Due Date (EDD) untuk penjadwalan produksi. *JMEIS, 2*(1), 65–72. https://doi.org/10.52330/jmeis.v2i1.267

Saha, N., Paul, P., Ji, K., & Harik, R. (2024). Performance evaluation framework of MQTT client libraries for IoT applications in manufacturing. *Manufacturing Letters, 41*, 1237–1245. https://doi.org/10.1016/j.mfglet.2024.09.150

Wang, C., Qiao, J., Huang, X., Song, S., Hou, H., Jiang, T., Rui, L., Wang, J., & Sun, J. (2023). Apache IoTDB: A time series database for IoT applications. *Proceedings of the ACM on Management of Data, 1*(2), Article 195. https://doi.org/10.1145/3589775

Wijayaningrum, V. N., & Wakhidah, R. (2023). Monitoring development board pada platform InfluxDB dan Grafana. *Jurnal Informatika dan Teknologi Informasi, 20*(1), 81–90.

Xie, S., & Tan, H. Z. (2021). Blur-readable two-dimensional barcode based on blur-invariant shape and geometric features. *International Journal of Advanced Robotic Systems, 18*(2). https://doi.org/10.1177/17298814211002613

Younis, N. K., Ahmed, M. R., Talab, A. W., & Ali, R. R. (2025). Testing real-time system algorithms performance on synthetic data: Analytical study of hard and soft system tasks. *International Journal of Innovative Research and Scientific Studies, 9*(5), 607–614. https://doi.org/10.55214/25768484.v9i5.6956

Zhang, Y. (2025). A dynamic scheduling method combining iterative optimization and deep reinforcement learning to solve sudden disturbance events in a flexible manufacturing process. *Mathematics, 13*(1), 4. https://doi.org/10.3390/math13010004

Zhao, A., Liu, Y., Zhang, Z., Wang, J., Cheng, Y., & Pan, X. (2022). Data-mining-based real-time optimization of the job shop scheduling problem. *Mathematics, 10*(23), 4608. https://doi.org/10.3390/math10234608

Zulfadlillah, Z., Fathani, M. A., Noval, M., & Irwana, I. (2024). Penerapan sistem Manufacturing 4.0 dengan integrasi Internet of Things (IoT) untuk optimalisasi efisiensi produksi. *Juritek, 4*(1), 159–164. https://doi.org/10.53625/juritek.v4i1.7234