УДК 004.6
DOI 10.17513/snt.40780

МОДЕЛИ ПРОГРАММНОГО УПРАВЛЕНИЯ СИСТЕМАМИ РАСПРЕДЕЛЕННОЙ ОБРАБОТКИ ДАННЫХ В КОМПЬЮТЕРНЫХ СЕТЯХ: СРАВНИТЕЛЬНЫЙ АНАЛИЗ И РАЗРАБОТКА ИНТЕГРАТИВНОЙ АРХИТЕКТУРЫ

Нугаев Р. К. ORCID ID 0009-0004-4049-9296, Сиразетдинов Р. Т.

Аккредитованное образовательное частное учреждение высшего образования «Московский финансово-юридический университет МФЮА», Москва, Российская Федерация, e-mail: rinat@nugaev.net

В статье рассматривается актуальная проблема программного управления системами распределенной обработки данных в условиях усложняющихся требований к адаптивности, гибкости соглашений об уровне обслуживания и поддержке гетерогенных ресурсов. Цель исследования – провести системный сравнительный анализ ключевых архитектур программного управления (MapReduce, Hadoop/YARN/HDFS, Spark, Flink, Kafka Streams) с учетом их преимуществ, ограничений и эксплуатационных метрик, а также обосновать целесообразность создания интегративной архитектурной модели. В работе использовались методы сравнительного анализа, моделирования эксплуатационных сценариев и формализации архитектурных решений с применением ориентированных ациклических графов, нагрузочного тестирования и статистической оценки эффективности. Результаты исследования показали, что классические и современные архитектуры имеют различные уровни гибкости, устойчивости и масштабируемости, при этом переход к гибридным, мультиагентным и оптимизируемым с применением методов машинного обучения моделям позволяет существенно повысить адаптивность, эффективность использования ресурсов и отказоустойчивость распределенных систем. Разработанная интегративная модель совмещает концепцию мультиагентного планировщика, динамический оркестратор процессов на основе направленных ациклических графов, кластеризацию задач с помощью алгоритмов машинного обучения и гибридное управление механизмами исполнения, что подтверждает эффективность предложенного подхода в типовых эксплуатационных сценариях. Сделан вывод, что дальнейшее развитие программного управления распределенной обработкой данных требует интеграции самонастраивающихся и интеллектуальных моделей, обеспечивающих высокий уровень надежности и адаптивности современных информационных платформ.

Ключевые слова: распределенные вычисления, программное управление, кластеризация, мультиагентные системы, оркестратор на основе направленного ациклического графа, потоковая обработка данных, Apache Spark, Apache Flink, отказоустойчивость, соглашение об уровне обслуживания, гибридная архитектура, прогнозирование нагрузки

SOFTWARE CONTROL MODELS FOR DISTRIBUTED DATA PROCESSING SYSTEMS IN COMPUTER NETWORKS: COMPARATIVE ANALYSIS AND DEVELOPMENT OF AN INTEGRATIVE ARCHITECTURE

Nugaev R. K. ORCID ID 0009-0004-4049-9296, Sirazetdinov R. T.

Accredited Private Higher Education Institution "Moscow Financial and Law University of the Moscow Financial and Law Academy", Moscow, Russian Federation, e-mail: rinat@nugaev.net

The article discusses the current problem of software management of distributed data processing systems in the context of increasingly complex requirements for adaptability, flexibility of service level agreements and support for heterogeneous resources. The purpose of the study is to conduct a systematic comparative analysis of key software management architectures (MapReduce, Hadoop/YARN/HDFS, Spark, Flink, Kafka Streams), taking into account their advantages, limitations, and operational metrics, as well as to substantiate the feasibility of creating an integrative architectural model. The work used methods of comparative analysis, modeling of operational scenarios and formalization of architectural solutions using oriented acyclic graphs, load testing and statistical efficiency assessment. The results of the study showed that classical and modern architectures have different levels of flexibility, resilience, and scalability, while the transition to hybrid, multi-agent, and machine learning-optimized models can significantly improve the adaptability, resource efficiency, and fault tolerance of distributed systems. The developed integrative model combines the concept of a multi-agent scheduler, a dynamic process orchestrator based on directed acyclic graphs, task clustering using machine learning algorithms and hybrid control of execution mechanisms, which confirms the effectiveness of the proposed approach in typical operational scenarios. It is concluded that the further development of software management of distributed data processing requires the integration of self-adjusting and intelligent models that ensure a high level of reliability and adaptability of modern information platforms.

Keywords: distributed computing, software management, clustering, multi-agent systems, directed acyclic graph orchestrator, streaming data processing, Apache Spark, Apache Flink, fault tolerance, service level agreement, hybrid architecture, load forecasting

Введение

В последние десятилетия вопросы эффективной организации и управления системами распределенной обработки данных в компьютерных сетях приобретают все большую значимость. Это связано с экспоненциальным ростом объемов информации, появлением облачных вычислений, интернета вещей и высокопроизводительных распределенных сервисов. Существенную роль в работе таких систем играет программное управление, от которого зависит координация вычислительных процессов, балансировка нагрузки, распределение заданий, а также поддержание устойчивости и надежности функционирования всей инфраструктуры. В современном программном обеспечении существует большое разнообразие архитектурных подходов: от централизованных моделей до децентрализованных и гибридных, а также решений, основанных на использовании контейнеризации и виртуализации.

Однако наличие широкого спектра технологических средств не устраняет комплекс нерешенных проблем. Множество существующих моделей страдают от ограниченной масштабируемости, сложностей синхронизации, высоких накладных расходов на координацию и низкой устойчивости к быстро меняющимся нагрузкам и отказам. Кроме того, внедрение новых технологических парадигм, таких как бессерверные вычисления, периферийные вычисления и глубокая контейнеризация, сопряжено с целым рядом вызовов при интеграции в традиционные схемы управления распределенной обработкой данных. Все это формирует потребность в новых универсальных и эффективных методах программного управления, способных быстро адаптироваться к преобразующимся условиям эксплуатации и обеспечивать высокие требования к качеству, надежности и производительности современных информационных систем.

Цель исследования – провести систематический и критический сравнительный анализ наиболее распространенных и широко применяемых моделей программного управления системой распределенной обработки данных в компьютерных сетях.

Материалы и методы исследования

Данное исследование проведено за период с 2025 по 2026 г. и основано на обобщении научных публикаций как зарубежных, так и отечественных авторов по теме о современных платформах распределенной обработки данных (РОД). В результате этого часть процессов была показана в виде схемы, на которой все этапы идут в заранее определенной последовательности, без циклов и возвратов назад. Благодаря такой структуре данных, которая называется направленный ациклический граф (DAG), можно управлять зависимостями и обеспечивать эффективное выполнение задач в компьютерных сетях. Эффективным инструментом анализа таких сложных систем является математическое моделирование, так как при этом появляется возможность представить различные сценарии нагрузочного тестирования, помогающие ответить на вопросы о границах производительности, о том, когда начинаются отказы и как система поведет себя под нагрузкой. Одновременно был осуществлен анализ влияния различных факторов на соблюдение соглашений об уровне обслуживания (Service Level Agreement, SLA) при устранении инцидентов, связанных с результатами, полученными в рамках моделируемых сценариев, что помогло определить уязвимые места при работе систем под нагрузкой.

Значительный вклад в методы исследования внес синтез пакетной и потоковой обработки входных данных в каждом узле системы РОД по мере их поступления, что позволило оценить взаимосвязь между архитектурными решениями и выявить потребность в более сложных гибридных вычислительных средах, которые ориентированы на повышение гибкости управления вычислительными процессами и более эффективное распределение ресурсов в распределенных средах. В рамках настоящего исследования в качестве такого подхода предложена новая интегративная модель (IAM), которая сочетает элементы как адаптивного планирования, так и динамической оркестрации вычислительных графов и методов интеллектуального анализа данных.

Результаты исследования и их обсуждение

На сегодняшний день, по мере того как объем, разнообразие и динамика данных усиливаются, а темпы изменений инфраструктуры стремительно возрастают, подходы к организации распределенных вычислений оказываются под влиянием не только технических, но и методологических критериев. Примечательно, что уже классические модели – например, модель параллельных вычислений MapReduce – в свое время представляли собой своего рода эталон надежности и масштабируемости, органично соответствующий задачам пакетной обработки, когда приоритетом служило сглаживание пиковых нагрузок за счет алгоритмов асинхронного исполнения и оптимизации

хранения [1]. Однако технологический спрос на снижение задержек и повышение интерактивности вычислений постепенно довел до осознания, что дальнейшее усложнение систем пакетной обработки данных не способно удовлетворить требованиям сценариев с обработкой данных в режиме реального времени.

В этом контексте становится наглядной разница между стабильностью, присущей традиционным системам, и растущей потребностью в адаптивности, которая свойственна архитектурам нового поколения. Неоднородность требований отражается – и весьма существенно – на выборе средств программного управления, что демонстрирует, с одной стороны, отход от централизованных решений, а с другой – стремление к интеграции разнообразных вычислительных парадигм. Рациональное инженерное мышление подталкивает к компактным, но функциональным моделям, где устойчивость, масштабируемость и гибкость балансируются не на словах, а средствами архитектурного проектирования.

Рассматривая современные примеры реализации подобных подходов, нельзя не отметить работу D. Zhang и соавт. [2], в которой разработанная для синхротронных лучевых линий схема обработки строится на основе интеграции распределенной файловой системы Hadoop Distributed File System (HDFS), диспетчера ресурсов Yet Another Resource Negotiator (YARN) и вычислительной платформы Apache Spark, дополненных микросервисным слоем для повышения модульности и обеспечения отказоустойчивости. Выбор этих технологий вовсе не случаен: зрелость, открытость и высокая степень стандартизации Apache Hadoop-экосистемы делают ее непременным инструментом для задач с большими объемами данных и повышенными требованиями к доступности, о чем справедливо отмечается и в публикации C. Ma и коллег [3]. Однако заметно, что ставка на увеличенную пропускную способность HDFS влечет за собой компромиссы по части консистентности и подходит лишь для специфических сценариев, прежде всего связанных с пакетной обработкой, тогда как в потоковых кейсах такая архитектурная упрощенность становится серьезным недостатком.

Централизованные механизмы управления ресурсами, воплощенные в YARN, на первый взгляд обещают простоту и прозрачность эксплуатации за счет централизованной логики диспетчеризации [4]. Однако, едва масштаб нагрузки выходит за пределы штатной эксплуатации, как показано в ряде исследований [5], подобная централизованность становится критической уязвимостью, снижая устойчивость общей системы к отказам. Возникает то напряжение между универсальностью архитектур и реальной управляемостью масштабируемых сред, которое невозможно разрешить без привлечения новых средств адаптации. На практике HBase, будучи распределенным хранилищем, формально наследует принципы BigTable и заявляет масштабируемость, тесно работает с Hadoop. Однако при ближайшем рассмотрении видно: устойчивость системы во многом условна. Стоит нагрузке стать неравномерной – появляются перегретые участки, отдельные зоны проседают по пропускной способности, и без кропотливого мониторинга с ручной корректировкой здесь не обойтись [2]. Получается, что даже развитые решения распределенного хранения все еще требуют регулярного вмешательства специалистов для поддержания стабильной работы. Если говорить шире, в программном управлении системами обработки данных происходит явный сдвиг – логика уже не жестко пакетная, а смешанная. Все больше появляется сценариев, где объединяются пакетные и потоковые вычисления. Spark в этом смысле показателен: механизм устойчивого распределенного набора данных (RDD) по сути связывает два подхода, позволяя строить решения по месту, исходя из реальных потребностей нагрузки и характера обработки. Согласно [6], RDD – это ключевой элемент архитектуры Spark, представляющий собой неизменяемую структуру данных, разделенную на секции, которые могут параллельно вычисляться и храниться как в памяти, так и на дисках различных узлов кластера. В Spark поддерживаются два типа операций над RDD: преобразования (например, map, filter, reduceByKey) и действия (count, first, сохранение результатов). Преобразования формируют новые RDD на базе существующих с отложенным выполнением, то есть вычисления запускаются только в момент обращения к действию, что позволяет строить гибкие и оптимальные вычислительные сценарии. Действия, в свою очередь, инициируют реальное выполнение всей цепочки зависимостей. Такая модель, объединяющая механизм отложенного выполнения и DAG, позволяет гибко адаптировать обработку к предметной специфике [6].

Типичная архитектура кластера Spark включает главный и несколько рабочих узлов, где каждый рабочий узел управляет памятью и распределением задач через модули-исполнители. Память исполнителя делится на области хранения (для кэширования и передачи данных) и выполнения (для

операций shuffle, aggregation, sort и др.). Ключевым параметром является конфигурация распределения памяти, настроек spark. executor. memory и spark.memoryFraction. Как показали эксперименты H. Li и соавт. [6], производительность Spark существенно зависит от пропорции памяти, выделенной под кэш, и сложности подбора оптимальной конфигурации под разные нагрузки. Например, при работе с массивом данных примерно в 70 ГБ (нагрузка Bayes, четыре узла в кластере) оказалось, что комбинация spark.memoryFraction = 0,6 и выделение по 15 ГБ на executor дала лучший результат. Казалось бы, если просто увеличить память до 30 ГБ, вычисления должны пойти быстрее, но на деле все вышло наоборот: время выросло с 2159 до 7904 с. Парадоксально, но факт. Более того, в [6] указывают, что максимальная производительность достигалась при других параметрах – все те же 30 ГБ, но с гораздо меньшим значением spark.memoryFraction, всего 0,1. Получается, система реагирует на смену настроек весьма остро. То, что один раз помогает, в другой ситуации мешает. Из-за этого возникает ощущение, что выбрать универсальные параметры невозможно – настройки ведут себя по-разному, нужна постоянная донастройка под задачу.

Это, видимо, и есть одна из основных трудностей работы с распределенной обработкой. Даже платформы, которые давно считаются стандартом в индустрии, как тот же Spark, требуют экспериментов – простая служебная инструкция тут не спасет, приходится разбираться с деталями каждый раз заново. К тому же сама логика Spark все-таки отличается от привычного подхода MapReduce. Здесь вычисления строятся через DAG – ориентированный ациклический граф, и меняется не только способ организации отдельных операций, но, пожалуй, вся структура процесса. Данные ее свойства позволяют реализовать на практике как пакетную, так и потоковую обработку с большим запасом параллелизма. Типичный пример – процесс исследования кристаллических структур с помощью Spark и DIALS [2; 7]. Тут возможности платформы раскрываются по-новому. При этом Spark демонстрирует высокую эффективность в параллельных расчетах и эксплуатации распределенных ресурсов для крупных аналитических задач [8]. Вместе с тем даже такая универсальная система, как Spark, обладает недостатками. В качестве примера можно привести ее модель обработки с использованием микропакетов (микробатчей), которая не в состоянии обеспечить минимальное время задержки для приложений, функционирующих в условиях обработки данных в реальном времени.

Постепенный переход к сценариям с малой задержкой привел к популярности иных платформ, сочетающих строгую семантику исполнения с поддержкой динамической маршрутизации потоков, таких как Flink и Kafka Streams. Исследование [9] фиксирует, что платформа Flink обеспечивает минимальные задержки передачи данных и в ряде интенсивных задач обработки потоков превосходит Spark, хотя при этом требует большего объема вычислительных ресурсов, сложнее в освоении и сопровождении, что ограничивает широкое внедрение в корпоративной среде. В свою очередь, Kafka Streams предлагает более легкую для интеграции модель управления потоковыми процессами, ориентированную на использование внутри экосистемы Apache Kafka и позволяющую находить баланс между надежностью работы и простотой администрирования, особенно востребованную в системах событийной обработки, где Kafka используется в качестве основного брокера данных [10].

Lambda-архитектура часто обсуждается именно потому, что здесь пытаются соединить пакетную и потоковую обработку в одной системе. По идее, так сохраняются сильные стороны обоих подходов. Но на практике объединение дает и немало трудностей: наряду с гибкостью растет и сложность, и нагрузка – причем как в плане организации процессов, так и в эксплуатации. В некоторых обзорах прямо отмечают, что сопровождать подобные решения тяжелее [1]. Чтобы как-то смягчить эти противоречия, появляются концепции более гибких архитектур, которые способны менять режим работы под нагрузку и масштабироваться автоматически. В литературе [11–13] есть такие примеры, хотя ощущение законченности в них отсутствует – это скорее поиски направления, чем устоявшаяся практика. Складывается впечатление, что из-за самой динамики вычислений приходится уходить от централизованных схем в пользу сложных, многоуровневых конструкций. При этом старые вопросы – устойчивость системы и надежность в реальных условиях – остаются, а иногда становятся даже более проблемными.

Похожие парадоксальные моменты видны и в эволюции микросервисной архитектуры. Сейчас разбивка крупных решений на относительно автономные сервисы меняет не только технику, но и весь подход к управлению инфраструктурой [14]. Да, масштабирование проще, локализация ошибок удобнее, появляется гибкость в распределении функций. Но с ростом числа сер-

висов неизбежно усложняются вызовы, взаимодействие между частями, координация процессов и контроль целостности данных. Особенно остро это проявляется уже в потоковых сценариях, где нельзя допускать задержек и ошибок согласованности. Дополнительный пласт сложности – программно-определяемые сети – Software-Defined Networking, SDN, и программно-определяемые центры обработки данных – Software-Defined Data Center, SDDC [15]. Направленность тут на гибкость, возможность простого управления, выделения логики от железа. Но централизованный контроллер, на котором все держится в SDN, иногда оказывается уязвимым местом – вполне достаточно всплеска нагрузки или сбоя, и есть риск нарушить работу всей системы.

Потому и обсуждаются сейчас переходные схемы: не совсем централизованные, но и не полностью распределенные. Часть работ [16] упирается именно в гибридность, организацию самоадаптирующихся контроллеров, хотя внедрение по-прежнему непростое – многие решения все еще на стадии прототипа, а масштабные внедрения буксуют из-за неразрешенных проблем с надежностью и управляемостью в боевых условиях.

Резюмируя рассмотренные подходы к программному управлению распределенной обработкой данных, целесообразно систематизировать их ключевые характеристики для наглядного сравнения (табл. 1).

Для объективного сопоставления рассмотрены эксплуатационные метрики в табл. 2.

Сравнительный анализ показывает, что выбор архитектуры всегда связан с особенностями прикладных задач, требованиями к задержкам, возможности масштабирования и допустимой сложностью эксплуатации. Тенденция развития современных систем – рост требований к адаптивности, гибкости управления SLA и поддержке гетерогенных ресурсов.

Таблица 1

Основные модели программного управления распределенной обработкой данных

Модель	Описание	Преимущества	Ограничения
MapReduce	Классическая модель пакетной обработки	Устойчивость, простота, масштабируемость	Высокая задержка, нет поддержки потоков
Hadoop + YARN + HDFS	Комплексная система пакетной обработки	Репликация, зрелость экосистемы	Ограниченная гибкость, проблемы с обработкой в реальном времени
HBase	Распределенное хранилище на HDFS	Масштабируемость, интеграция с Hadoop	Горячие регионы, высокая сложность конфигурации
Spark	Универсальный механизм для пакетной и потоковой обработки	DAG, устойчивость, интеграция с ML	Модель микропакетов → задержки, высокое потребление ресурсов
Apache Flink	Платформа потоковой обработки с точной семантикой (Exactly-once)	Низкая задержка, гибкость	Высокие ресурсы, сложность отладки
Kafka Streams / Samza	Потоковая обработка на основе системы обмена сообщениями Apache Kafka	Интеграция, отказоустойчивость	Ограниченная логика, зависимость от Kafka
Lambda-архитектура	Гибридная модель (пакетная + потоковая обработка)	Баланс между скоростью и консистентностью	Дублирование логики, сложности тестирования и поддержки

Примечание: составлена автором на основе полученных данных в ходе исследования

Таблица 2

Основные метрики разных архитектур

Модель	Время отклика (ms)	Масштабируемость	Отказоустойчивость	Оптимизация использования ресурсов
MapReduce	1000–30000	Средняя	Средняя	Низкая
Spark	200–10000	Высокая	Высокая	Высокая
Flink	100–3000	Высокая	Очень высокая	Средняя/высокая
Гибридные	зависят от конфигурации	Очень высокая	Очень высокая	Очень высокая

Примечание: составлена автором на основе источников [14, 15]

В русле этих размышлений представляется обоснованным предложение интегративной модели IAM, построенной на следующих инновационных идеях:

1. Иерархический мультиагентный планировщик, который распределяет вычислительные задачи по уровням инфраструктуры – от периферийных устройств до центральных узлов – с учетом текущей загрузки и структуры кластера.

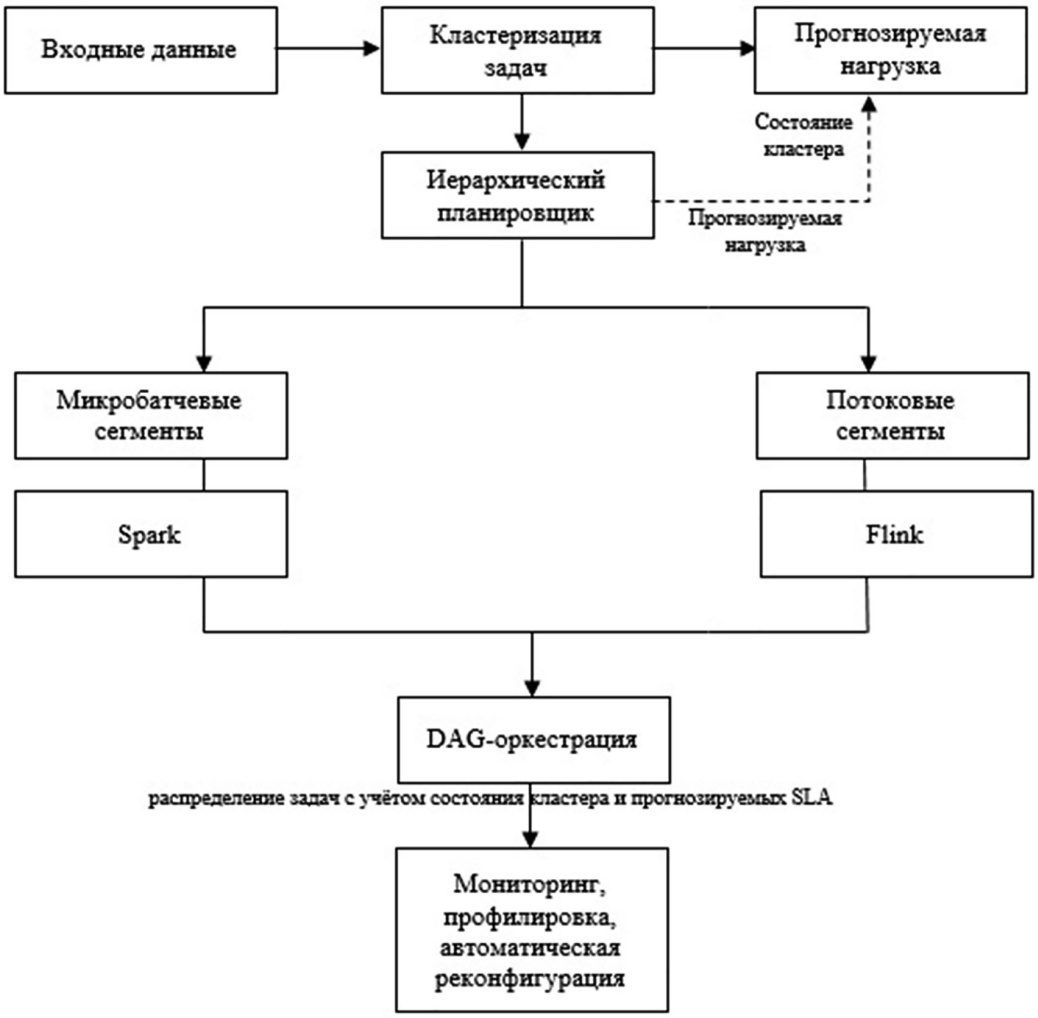
2. DAG-оркестратор с динамической перестройкой, формирующий вычислительный граф исполнения задач на основе анализа нагрузки, характеристик доступных ресурсов, параметров SLA и статистики работы системы в реальном времени.

3. Модуль кластеризации входных событий использует алгоритмы машинного обучения (например, HDBSCAN, OPTICS) для сегментации входных задач по профессиональным и содержательным признакам, определяя группы приоритета и пути обработки.

4. Система гибридной интеграции стримовых и пакетных исполнительных механизмов, позволяющая комбинировать работу Spark Structured Streaming и Apache Flink для поддержки задач с различными требованиями к задержке и особенностями жизненного цикла.

5. Блок прогнозирования нагрузки, построенный, в частности, на технологиях градиентного бустинга и ансамблей деревьев решений, предоставляющий возможность оценивать предстоящий поток вычислительных запросов и распределять ресурсы с учетом прогнозируемого профиля нагрузки.

Иерархическая структура IAM наглядно показана на рисунке, где архитектурный уровень определяется мультиагентным подходом, динамикой DAG-графов и активным использованием ML-компонент для поддержки SLA и механизмов самооптимизации.



Блок-схема оригинальной интегративной архитектуры программного управления
Примечание: составлен автором по результатам данного исследования

Для математического обоснования предложенной архитектуры используется представление всей системы в виде ориентированного ациклического графа G = (V,E), где множество вершин $V = \{v_i \mid i = 1,\ldots,|V|\}$ соответствует совокупности вычислительных задач, а множество ребер $E \subseteq V \times V$ отражает зависимости их выполнения, а также потоки данных и управляющих воздействий между задачами.

Исполнение задач осуществляется на множестве вычислительных и управляющих узлов $N = \{n_j \mid j = 1,\ldots,|N|\}$, формирующих распределенную инфраструктуру обработки. Таким образом, каждая задача $v_i \in V$ в процессе функционирования системы должна быть сопоставлена одному из узлов $n_j \in N$, что формализуется далее с помощью отображения назначения $A : V \to N$, где $A(v_i) = n_j$ означает, что задача v_i назначена для выполнения на узле n_j. Отображение A рассматривается как функция выбора из множества всех допустимых отображений, удовлетворяющих ресурсным, временным и логическим ограничениям системы.

Для каждой задачи $v_i \in V$ определяются:

− требуемое количество ресурсов R_i, характеризующее совокупное потребление вычислительных и системных ресурсов;

− прогнозное время выполнения на узле n_j, обозначаемое как $\hat{t}(v_i, n_j)$, учитывающее вычислительные характеристики узла и сетевые задержки;

− политика обработки, задаваемая функцией $f : V \to Z^+$, где Z^+ – множество положительных целых чисел, интерпретируемых как уровни приоритета или классы обслуживания задач в зависимости от текущего состояния системы и требований SLA.

С учетом структуры графа G и введенных параметров задача распределения формулируется как задача минимизации максимального времени выполнения вдоль критических путей графа:

$$\min_{A} \max_{p \in P(G)} \sum_{v_i \in P} \hat{t}\left(v_i, A\left(v_i\right)\right), \quad (1)$$

где P(G) – множество всех допустимых путей от начальных вершин (истоков) графа к конечным вершинам (стокам). Данная постановка отражает необходимость минимизации длительности выполнения наиболее протяженных по времени цепочек зависимых задач, определяющих общее время завершения вычислительного процесса.

Формулировка задачи начинается с набора ограничений, но уже на этом этапе видно: они не изолированы, а часто наслаиваются, в итоге сужая область допустимых решений.

Первое и самое очевидное ограничение – ресурсы на узлах. Система должна не допускать, чтобы совокупный спрос задач превысил их объем:

$$\sum_{v_i \in V_j} R_i \le C_j, \forall n_j \in N, \quad (2)$$

где $V_j = \left\{ v_i \in V \mid A\left(v_i\right) = n_j \right\}$ – множество задач, назначенных узлу n_j;

C_j – доступная емкость узла n_j.

На первый взгляд это выглядит просто, но для распределенных систем с их изменчивостью и неоднородностью ресурсов граница между допустимым и недопустимым становится плавающей. К этому добавляются зависимости между задачами – порядок выполнения зачастую фиксируется явно:

$$\text{start}(v_k) \ge \text{end}(v_i), \ \forall \left(v_i \to v_k\right) \in E. \quad (3)$$

Далеко не всегда именно ресурсы ограничивают расписание: бывает, что порядок задач решает больше.

Другая группа ограничений – требования SLA. Для каждого задания определяется предел по времени:

$$\forall v_i \in V : \ \hat{t}\left(v_i, \ A\left(v_i\right)\right) \le \text{SLA}_i. \quad (4)$$

Здесь формулируется новый конфликт – уложиться в SLA и одновременно не перегружать узлы не всегда возможно, и уже система должна выбирать компромисс между этими условиями.

Все это подводит к очевидному выводу: задача быстро набирает сложность и требуется способ, сокращающий перебор различных вариантов решений с помощью подсознательного мышления и интуиции, – эвристические нешаблонные оригинальные процедуры на основе взвешенного распределения, в котором используется ранжирование узлов по таким критериям, как:

− производительность (в основном задержка и пропускная способность во время предсказания) модели,

− доступная емкость узла,

− статистические данные за прошлые выполнения.

И тут важно понять назначение предлагаемой модели IAM. Речь идет о средах, где устойчивые состояния скорее редкость, чем норма. В распределенных инфраструктурах – будь то гибридные облачные конфигурации, периферийные сегменты или мультидоменные системы – параметры среды меняются быстрее, чем их успевают фиксировать классические методы управления.

Таблица 3

Ключевые критерии оценки эффективности архитектуры

Критерий	Описание	Алгоритм расчета	Градация оценки
Гибкость обработки	Характеризует способность системы перераспределять ресурсы между задачами с различными требованиями SLA	$S_{flex} = \Delta R / \Delta t$, где ΔR – объем перераспределенных ресурсов, Δt – время реакции на изменение требований	$S_{flex} \geq 0,9$ – высокая гибкость; $0,6 \leq S_{flex} < 0,9$ – средняя гибкость; $S_{flex} < 0,6$ – низкая гибкость.
Время отклика	Оценивает как средние, так и пиковые характеристики выполнения	$S_{resp} = \frac{1 - (avg(RTT) - (RTT\_ref))}{RTT_{ref}}$, где avg (RTT) – среднее время отклика, RTT_{ref} – эталонное (референсное) время	$S_{resp} \geq 0,95$ – отличное; $0,90 \leq S_{resp} < 0,95$ – хорошее; $S_{resp} < 0,90$ – удовлетворительное.
Эффективность планирования	Отражает степень отклонения от условно оптимального распределения задач	$S_{plan} = \frac{1 - \sigma(planopt - planreal)}{planopt}$, где σ – стандартное отклонение (по всем задачам), planopt – оптимальное распределение, planreal – реальное распределение	$S_{plan} \geq 0,95$ – эффективное $0,8 \leq S_{plan} < 0,95$ – приемлемое $S_{plan} < 0,8$ – неудовлетворительное
Устойчивость к нагрузке	Фиксирует изменение пропускной способности при росте нагрузки	$Sload = - dThroughput / dLoad$ где dThroughput – изменение пропускной способности, dLoad – изменение нагрузки	$Sload \geq 0$ – устойчива $Sload < 0$ – деградирующая

Примечание: составлена автором на основе полученных данных в ходе исследования

В этих условиях попытка добиться строгой оптимальности оказывается менее продуктивной, чем ориентация на адаптивность, пусть даже с потерей некоторой точности. По сути, приоритет смещается: важнее сохранить работоспособность и согласованность системы, чем добиться формально наилучшего решения в каждый момент времени.

Проверка жизнеспособности предложенного подхода, исходя из этого, также требует несколько иной постановки. Недостаточно ограничиться сравнением с альтернативными решениями в фиксированных условиях. Предполагается использование моделирования, в котором варьируются как нагрузка, так и доступные ресурсы, причем в динамике. Это позволяет наблюдать не только конечные состояния, но и сам процесс адаптации – те переходные режимы, где и проявляются основные преимущества или, напротив, ограничения архитектуры.

Основная цель – зафиксировать улучшения по ряду критериев, представленных в табл. 3.

Наличие градаций для каждого критерия позволяет не столько получить итоговую оценку, сколько выявить слабые места – иногда это оказывается более информативным, чем единый агрегированный показатель.

Тем не менее для обобщенной интерпретации используется интегральная метрика:

$$S_{total} = w_1 * S_{flex} + w_2 * S_{resp} + w_3 * S_{plan} + w_4 * S_{load}, \quad (5)$$

где веса w_k нормированы ($\sum_{k=1}^{4} w_k = 1,\ w_k \geq 0$) и отражают относительную значимость критериев. При этом сами веса не являются фиксированными – их выбор в значительной степени зависит от приоритетов конкретной системы. В одних случаях критичным становится время отклика, в других – устойчивость или гибкость, и это неизбежно влияет на итоговую оценку.

Определение весов осуществляется на основе комбинированного подхода, включающего экспертную оценку, формализуемую, в частности, с использованием метода анализа иерархий, а также эмпири-

ческую калибровку по результатам моделирования и анализа функционирования системы. Дополнительно проводится анализ чувствительности интегрального показателя к вариациям весовых коэффициентов, что позволяет оценить устойчивость выводов и снизить влияние субъективного фактора при их задании.

Для интерпретации интегрального показателя с целью оценки существующего уровня эффективности архитектуры разработана следующая шкала эффективности:

– St_{otal} в пределах 0,90 и выше – уровень эффективности отличный;

– St_{otal} в пределах 0,80–0,89 – уровень эффективности высокий;

– St_{otal} в пределах 0,70–0,79 – уровень эффективности хороший;

– St_{otal} в пределах 0,60–0,69 – уровень эффективности удовлетворительный;

– St_{otal} ниже 0,60 – уровень эффективности неудовлетворительный.

Предложенный методологический подход дает двойной эффект: с одной стороны, не теряются детали, с другой – итоговые выводы легче интерпретировать и их проще проследить до конкретных причин. Такой подход пригоден не только для исследований, но и для практических задач – например, аудита или оптимизации распределенных вычислительных систем, где важна прозрачность оценки.

Для обеспечения воспроизводимости и релевантности анализа предполагается использовать следующие среды и инструменты (табл. 4).

План верификационного эксперимента включает проектирование нагрузочных сценариев для проверки основных гипотез (табл. 5).

Таблица 4

Инструменты и компоненты экспериментальной среды

Категория	Инструмент / Технология	Назначение
Виртуализация и кластеризация	Kubernetes, Docker Swarm	Автоматическое масштабирование, отказоустойчивость
Движки обработки	Apache Spark, Apache Flink, IAM (предложенная архитектура)	Исполнение пакетных, потоковых и гибридных задач
Мониторинг	Prometheus + Grafana	Сбор и визуализация метрик производительности
Генерация нагрузки	Apache JMeter, Locust	Моделирование событий: пакетные и потоковые сценарии
Профилировка и трассировка	cAdvisor, Jaeger	Отслеживание использования ресурсов и критического пути DAG
Формальная симуляция (при необходимости)	OMNeT++, NS3	Моделирование сетевых характеристик среды
ML-библиотеки для прогнозирования и кластеризации	scikit-learn (HDBSCAN, GBDT)	Сегментация событий и предсказание нагрузки

Примечание: составлена автором на основе полученных данных в ходе исследования

Таблица 5

Сценарии нагрузочного тестирования

№	Сценарий	Цель	Ключевые метрики
1	Базовая нагрузка	Сравнение IAM с Spark и Flink при стандартной нагрузке (пакетной и потоковой)	avg(RTT), σ (plan_opt – real), CPU/Memory Util
2	Всплеск нагрузки	Изучение адаптации IAM к внезапным изменениям входного потока	dThroughput/dLoad, SLA violations
3	Ограниченные ресурсы	Проверка устойчивости IAM при дефиците ресурсов	Плавное снижение производительности (graceful degradation), avg(RTT)
4	Переключение SLA-политик	Оценка гибкости DAG-оркестратора и планировщика при изменении SLA	ΔR/Δt, SLA Fulfillment Rate
5	Горячие зоны	Проверка равномерности распределения задач	Дисбаланс нагрузки процессора (CPU imbalance), отклонение распределения ресурсов (R_i deviation)

Примечание: составлена автором на основе полученных данных в ходе исследования

Формализация процедур подразумевает:

– использование унифицированных YAML-профилей нагрузки (типовая генерация Locust/JMeter);

– регистрация и экспорт метрик средствами Prometheus, их обработка через pandas (Python);

– формальное описание DAG-структур, критического пути и отклонения от оптимального планирования через NetworkX;

– идентичные настройки (лимиты, kwarg) для сравниваемых архитектур;

– статистическая обработка: усреднение по не менее 10 прогонов, оценка доверительных интервалов, t-тесты для подтверждения статистической значимости различий.

Проведенный анализ, а также результаты первичного моделирования в целом позволяют говорить о том, что архитектура IAM действительно усиливает гибкость распределения ресурсов между задачами с различными требованиями по SLA. Однако эта гибкость возникает не сама по себе, а формируется за счет сочетания нескольких механизмов, каждый из которых вносит свой вклад – не всегда равномерный и не во всех режимах одинаково значимый. Например, симбиоз иерархического планировщика с механизмом динамической перестройки графов вычислений позволяет обойтись без постоянных ручных корректировок. Прогнозирование активных нагрузок на основе кластеризации данных способствует сокращению перегруженных участков, которые обычно являются источником каскадных задержек. Это заметно в тех задачах, которые особо чувствительны к параметрам времени отклика, включая периферийные процессы реального времени, где даже небольшие отклонения от нормы способны в отдаленной перспективе спровоцировать значительные проблемы, так как имеют свойство накапливаться. Кроме того, мультиагентная многоуровневая организация лучше адаптируется к ситуациям, возникающим в реальной динамической среде, что положительно отражается на отказоустойчивости в системах РОД за счет перераспределения рисков, пусть даже это сопровождается усложнением координации. И немаловажно, что использование компонентной модели, в основе которой независимые модули, которые взаимодействуют между собой через интерфейсы и API, делает архитектуру менее жесткой, что в долгосрочной перспективе облегчает адаптацию системы к меняющимся требованиям – что, собственно, и становится одним из ключевых условий ее применимости.

В то же время IAM предъявляет высокие требования:

– к квалификации команды (настройка DAG, интеграция ML-компонентов),

– вычислительным ресурсам (расчеты кластеризации и прогнозирования в реальном времени),

– надежности мониторинга (ошибки в предсказании могут привести к рассогласованию графа).

Кроме этого, на текущем этапе работа ограничивается архитектурным и методическим анализом, а проведение полномасштабных эмпирических экспериментов с реализацией всех нагрузочных сценариев определено как задача дальнейших исследований. Практическая реализация и эмпирическая апробация IAM представляют особый интерес для сфер, в которых актуальны:

– быстрая реакция на события и гибкое распределение ресурсов – обработка IoT-данных с высокой изменчивостью (умные города, транспорт, энергетика, телекоммуникации);

– научные симуляции с жесткими SLA;

– гибридные облака, а также финансовые сервисы с высоким приоритетом скорости обработки и предотвращения мошенничества.

В целом результаты позволяют говорить о том, что у адаптивных распределенных платформ есть потенциал для дальнейшего развития – причем не только за счет отдельных технических доработок, но и благодаря способности самой системы подстраиваться под перемены, как внешние, так и внутренние.

Заключение

Проанализированные здесь подходы к программному управлению распределенными вычислениями показывают довольно четкое движение в сторону интеграции разных архитектурных решений – автоматизация, самоадаптация, многозадачность, использование ML. Модель IAM на этом фоне выглядит более эффективной по ряду эксплуатационных показателей и, что важно, хорошо вписывается в современные требования к гибкости и масштабируемости.

Практическая ценность во многом связана с тем, что вопросы управления сложными, быстро меняющимися сервисами сегодня стоят особенно остро. Видно, что IAM здесь может дать ощутимый эффект. В качестве дальнейших перспектив, очевидно, потребуется больше эмпирических проверок, нужна будет работа на реальных промышленных сценариях и с реальными нагрузками – только так удастся окончательно подтвердить жизнеспособность предложенного подхода.

Список литературы

1. Vilarrodona C. R.-C., Alvarez P., Lordan F., Alvarez J., Ejarque J., Badia R. M. A survey on the Distributed Computing stack // Computer Science Review. 2021. Vol. 42. Is. 2. 100422. DOI: 10.1016/j.cosrev.2021.100422.

2. Zhang D., Dai Z.-Y., Sun X.-P., Wu X.-T., Li H., Tang L., He J.-H. A distributed data processing scheme based on Hadoop for synchrotron radiation experiments // Journal of Synchrotron Radiation. 2024. Vol. 31. P. 635–645. DOI: 10.1107/S1600577524002637.

3. Ma C., Zhao M., Zhao Y. An overview of Hadoop applications in transportation big data // Journal of traffic and transportation engineering (English edition). 2023. Vol. 10. Is. 5. P. 900–917. DOI: 10.1016/j.jtte.2023.05.003.

4. Liu Y., Zeng Y. K., Piao X. F. High-Responsive Scheduling with MapReduce Performance Prediction on Hadoop YARN // Proceedings of the 2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 17–19 August 2016, Daegu, South Korea. P. 238–247. DOI: 10.1109/RTCSA.2016.51.

5. Yao Y., Gao H., Wang J., Sheng B., Mi N. New Scheduling Algorithms for Improving Performance and Resource Utilization in Hadoop YARN Clusters // IEEE Transactions on Cloud Computing. 2021. Vol. 9. Is. 3. P. 1158–1171. DOI: 10.1109/TCC.2019.2894779.

6. Li H., Ji S., Zhong H. et al. LPW: an efficient data-aware cache replacement strategy for Apache Spark // Science China Information Sciences. 2023. Vol. 66. Is. 112104. DOI: 10.1007/s11432-021-3406-5.

7. Winter G., Waterman D., Parkhurst J. et al. DIALS: implementation and evaluation of a new integration package // Acta Crystallographica Section D: Structural Biology. 2018. Vol. 74. Is. 2. P. 85–97. DOI: 10.1107/S2059798317017235.

8. You Z., Hu H., Wang Y., Xue J., Yi X. Improved Hybrid Collaborative Fitering Algorithm Based on Spark Platform // Wuhan University Journal of Natural Sciences. 2023. Vol. 28. Is. 5. P. 451–460. DOI: 10.1051/wujns/2023285451.

9. Ullah F., Dhingra S., Xia X., Ali Babar M. Evaluation of Distributed Data Processing Frameworks in Hybrid Clouds // Journal of Network and Computer Applications. 2024. Vol. 224. P. 1–14. DOI: 10.1016/j.jnca.2024.103837.

10. Šprem Š., Tomažin N., Matečić J., Horvat M. Building Advanced Web Applications Using Data Ingestion and Data Processing Tools. Electronics. 2024. Vol. 13. Is. 4. P. 1–23. DOI: 10.3390/electronics13040709.

11. Mezati M., Aouria I. Machine learning in big data: A performance benchmarking study of Flink-ML and Spark ML-lib // Applied Computer Science. 2025. Vol. 21. Is. 2. P. 18–27. DOI: 10.35784/acs_7297.

12. Ramisetty S., Chandrasekaran T., Eruvaram V. K., Pulicharla M. R. Optimizing Real-Time Data Pipelines for Machine Learning: A Comparative Study of Stream Processing Architectures // World Journal of Advanced Research and Reviews. 2024. Vol. 23. Is. 03. P. 1653–1660. DOI: 10.30574/wjarr.2024.23.3.2818.

13. Henning S., Hasselbring W. Benchmarking scalability of stream processing frameworks deployed as microservices in the cloud // Journal of Systems and Software. 2024. Vol. 208. 17 p. DOI: 10.1016/j.jss.2023.111879.

14. Song Y. F., Li C., Xuan K., et al. Automatic data archiving and visualization at HLS-II // Nuclear Science and Techniques. 2018. Vol. 29. Is. 9. P. 129. DOI: 10.1007/s41365-018-0461-6.

15. Aljughaiman A., Almarri S. The pivotal role of software defined networks to safeguard against cyber-attacks: a comprehensive review // PeerJ Comput Sci. 2025. Vol. 11. P. e2814. DOI: 10.7717/peerj-cs.2814. PMID: 40567704; PMCID: PMC12190307.

16. Alsheikh R., Fadel E., Akkari N. Distributed Software-Defined Networking Management: An Overview and Open Challenges // Aro-The Scientific Journal of Koya University. 2024. Vol. 12. Is. 2. P. 157–166. DOI: 10.14500/aro.11468.

Конфликт интересов: Авторы заявляют об отсутствии конфликта интересов.

Conflict of interest: The authors declare that there is no conflict of interest.

Финансирование: Авторы заявляют об отсутствии внешнего финансирования.

Financing: The research was performed without external funding.