Тройка нефункциональных требований:
Производительность, **надежность**, **безопасность**

Всегда компромисс. Нельзя всё сразу.
Но этой тройкой не исчерпываются нефункциональные требования.
Добавляем **масштабируемость и универсальность**.

Сложность корпоративной информационной системы в том, что она очень тесно интегрируется в бизнес-процесс предприятия, и это не та вещь, которую може поменять относительно быстро. Инфраструктуру менять можно (например, перейти с win 8 на 10). Сменить с одной КИС на другую за 1 день == сломать бизнес целиком. Когда система ставится, мы ждем, что она будет масштабируема: расширение бизнеаса, перепрофилирование, изменение конъюктуры рынка не должно привести к необходимости менять информационную систему предприятия. Масштабируемость тоже вступает в конфликт с производительностью/надежностью/безопасностью.

Обратная совместимость с предыдущими версиями
Это бич во всех ИС и вообще во всём ПО. ОС имеют сложную архитектуру для того, чтобы обеспечить обратную совместимость.

Внешняя совместимость с АО и ПО
Нужно построить систему, которая будет работать на какой-то системе на каком-то железе, при этом по-максимуму использовать возможности для реализации своих задач.

Соответствие отраслевым или другим стандартам / соответствие законодательству
(например, медицинские ИС)

Приходим к типовым решениям.
Меташаблоны решения в архитектуре систем — уже с набором компромиссов и дальше можно их развивать. Обычно все базируются на клиент-сервености (отделение хранения/обработки данных от представления)

Выделим три слоя в изначальной функциональной архитектуре [Любая информационная система реализует 5 информационных процессов: сбор, обработка, хранение, передача и представление информации. Любые операции в ИС комбинации этих функций. На]:
— Слой представлений (решает задачи сбора+представления —это интерфейс)
— Слой бизнес-логики (решает задачу обработки)
— Слой данных (решает задачу хранения)
Передача размазана между ними, обеспечивая взаимодействие слоев между собой.

Представления	Интерфейс								
Бизнес-Логика	Обработка								
Данные	Клиент доступ к данным	Файл-сервер Хранение данных							

Проблема с безопасностью. Максимум — ограничение доступа на уровне ФС, либо отсутствие разграничения
Проблема с атомарностью/параллельной обработки
…
Единственный плюс — дешивизна и простота

Второй мета-шаблон — классический/двузвенный клиент-сервер

Представления	Интерфейс								
Бизнес-Логика	Обработка	Обработка							
Данные	Клиент	Сервер Доступ и хранение данных							

Невозможно создать систему, где вся обработка на сервере или вся обработка на клиенте.

Для толстого клиента (обработка приемущественно на клиенте) недостаток — высокая сложность администрирования и настройки клиентских рабочих мест. Сложно производить обновления. Требует затрат на аппаратное обеспечение рабочих мест.

Для тонкого клиента (обработка приемущественно на сервере) требуется стабильность и высокая пропускная способность. Проблема нагрузки на сервера.

Используя хранимые процедуры повышаем безопасность. Не знаем сведения о реальном хранении данных.
Если надо что-то изменить, то всё быстро и безопасно (транзакции в базу. блокируют немного и ненадолго)

Переходим к трехзвенной архитектуре

Представления	Интерфейс								
Бизнес-Логика	Обработка	Обработка							
Данные	Клиент	Сервер приложений	Доступ и хранение данных Сервер данных						

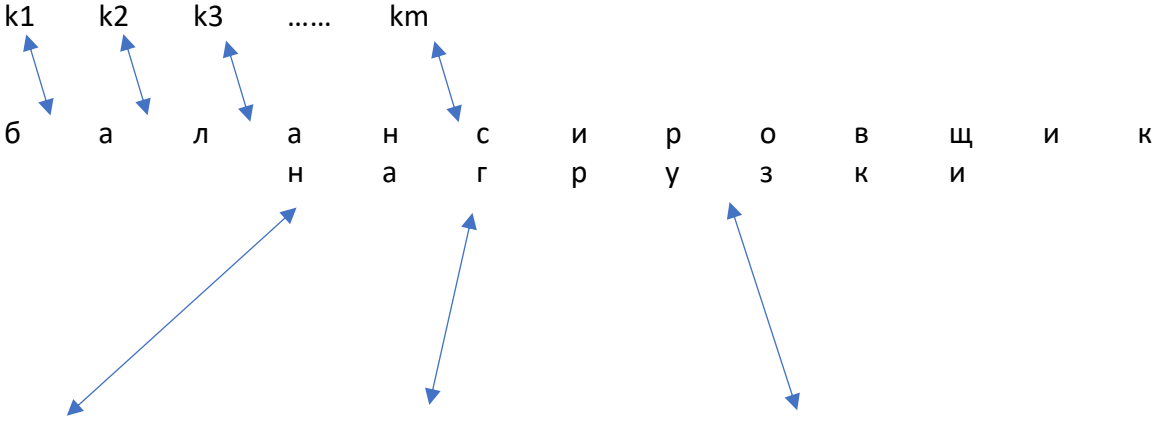
Заточено под тонкие клиенты. Простая нативная реализация тонкого клиента, минимизация потока между клиентом и сервером приложений. Нативная высокая безопасность. Минимизирован поток между клиентом и сервером приложений (упаковываем данные).
Можно обработку продублировать:

Представления	Интерфейс								
Бизнес-Логика	Обработка	Обработка 1 Обработка 2 ……. Обработка k							
Данные	Клиент	Сервер приложений	Доступ и хранение данных Сервер данных						

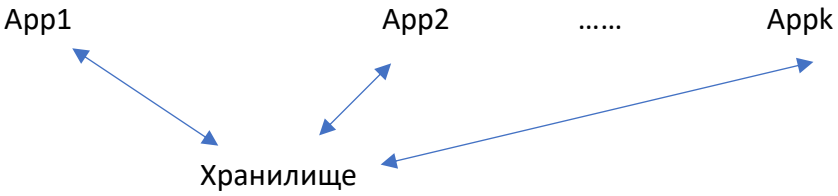
Сервер данных в отдельной виртуальной сети, что повышает безопасность. Всё ещё подразумевается строго централизованное хранилище данных.

Теория массового обслуживания {queuing system}: когда много клиентов и много обработчиков.
Есть требования. Например, web-server. Требования — http(s) запросы. Есть приборы (сервера приложнний), которые их формируют.
всё работает хорошо когда поток пуассоновский (много источников требований (->inf), но при этом требования они дают довольно редко — тогда в целом наиболее вероятна ситуация что в зависимости от количества источников требования движутся с каким-то интервалом (есть наиболее вероятный интервал). Может возникать [с меньшей вероятностью] ситуация больших сгущений. С другой стороны, наоборот, может возникать какая-то пауза. В Пуассонвском процессе возникают задачи резкого наростания/спада), а обработчики с минимально меняющейся производительностью.

Надо строить те или иные виды балансировки для управления очередью входящих требований
Мы не можем предугатать сколько тот или иной сервер будет выполнять запрос. Разное железо, разная загруженность и так
Синхронная балансировка (например настройки nginx)
клиенты:



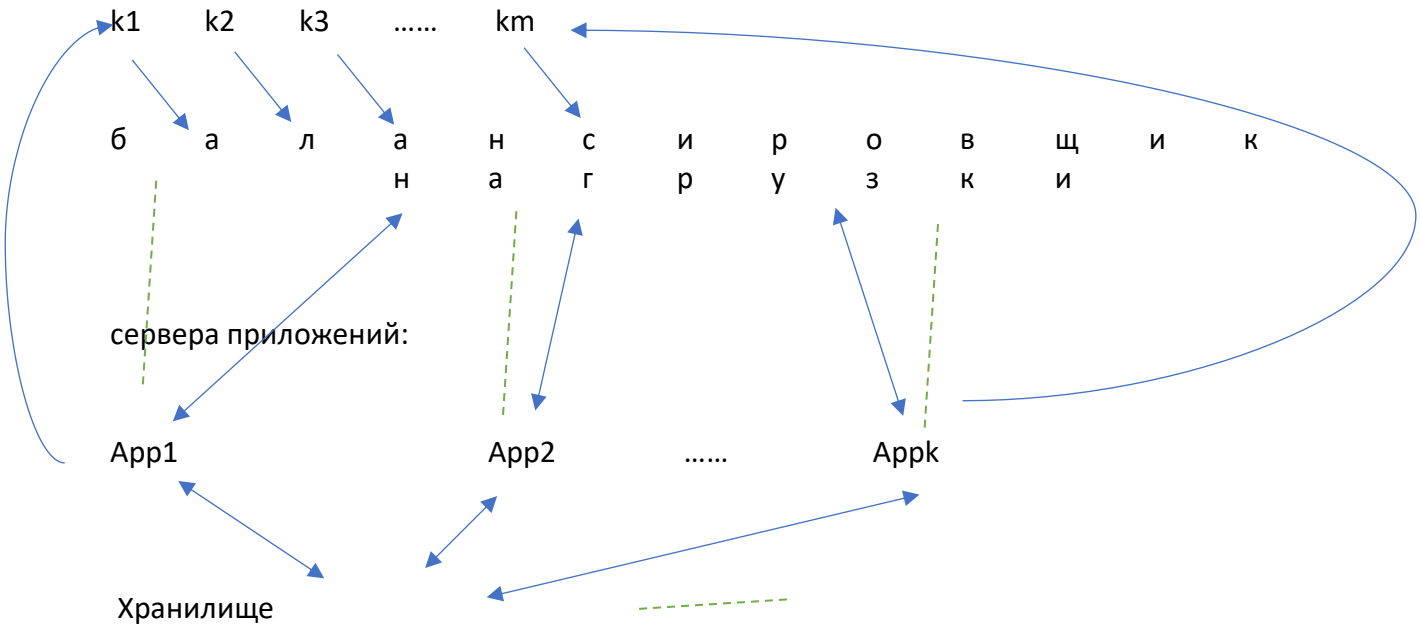
сервера приложений:



При синхронной балансировке и входной поток и выходной поток происходит через балансировщик. Это позволяет очень четко отследить состояние каждого сервера. Балансировщик знает (я его отправил и столько требований было. Сколько ответов получил и тд) у какого сервера сколько сейчас в обработке. Знаем кто вышел из строя (не получаем ответов). Хороший балансировщик может кешировать запросы, и, если ответа нет, отправлять на другой сервер. Удалять кеш только при получении ответа.
Можем очень эффикитивно использовать ресурсы серверов приложений.

НО: возникает проблема безопасности (вскрыли балансировщик -> пролезть дальше), проблема одной точки входа: умер балансировщик -> умерло всё.
Узким можно назвать канал балансировщика на отправку. Входной поток сильно меньше выходного.

Есть **ассинхронная балансиовка.**



Но балансировщик не может проверить, насколько всё ок работает. Он что-то отправляет как в черный ящик и ничего не знаем. Вводим систему мониторинга с своими приложениями (балансировщик опрашивает сервера: живы ли они, всё ли хорошо, сколько свободно памяти, сколько сейчас висит процессов в обработке). Но попадаем в

проблему мониторинга: либо мы будем мониторингом нагружать систему, либо иметь недостоверные данные.
Решение:
Многоагентные системы. На каждом сервере агент, который следит за состоянием своего сервера. У него есть некие требования (пороговые значения) для реакции. Если что он сам сообщит это балансировщику. Балансировщик либо снизит нагрузку, либо скажет держитесь там. Всё ещё остается проблема централизации. Выход из строя балансировщика нарушит работу системы.
Третий путь (не классическая балансировка).
Использование общей (управляемой) очереди. Кролик (RabbitMQ) обслуживает очередь. с одной стороны приходят запросы от клиентов, с другой контракты со стороны серверов (сервер который свободен кидает туда возможность выполнить тот или иной запрос). Задача очереди организовать встречу клиента и контракта. Неплохой вариант, но в этой ситуации сложно утилизировать ресурсы (как сервер должен понять что он может выполнить запрос — сложно). Сложно гарантировать время отклика (может возникнуть ситуация голодания, например) — нет механизма управления очередью (а если писать сложные механизмы обработки, то мы просто напишем балансировки).
Можно писать многоуровневые/универсальные очереди — но чем более сложный алгоритм — тем больше накладные ресурсы на реализацию этого алгоритма.

И в случае и синхронной, и асинхронной балансировки, пула ресурсов имеем общее хранилище данных.
хранилище становится узким местом.

Нужно разделить хранилище, одновременно обеспечив целостность данных: перейти к **распределенной системе**.