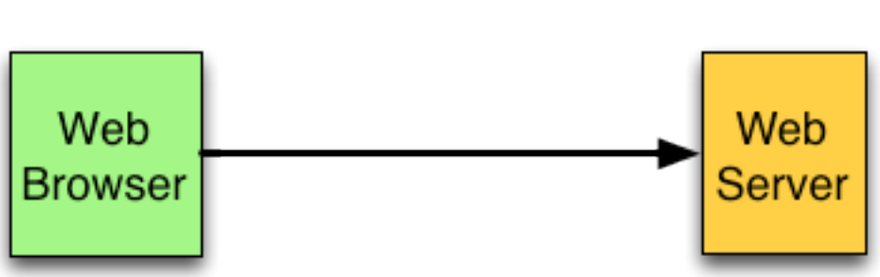
Finding the Needle in the Haystack (Troubleshooting Distributed Systems)

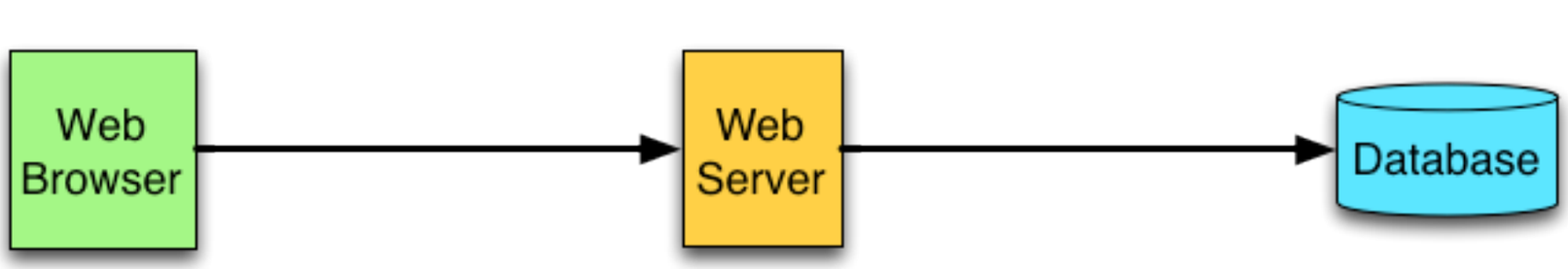
Anthony Molinaro
Erlang Factory 2014

Web Services have gotten more complex

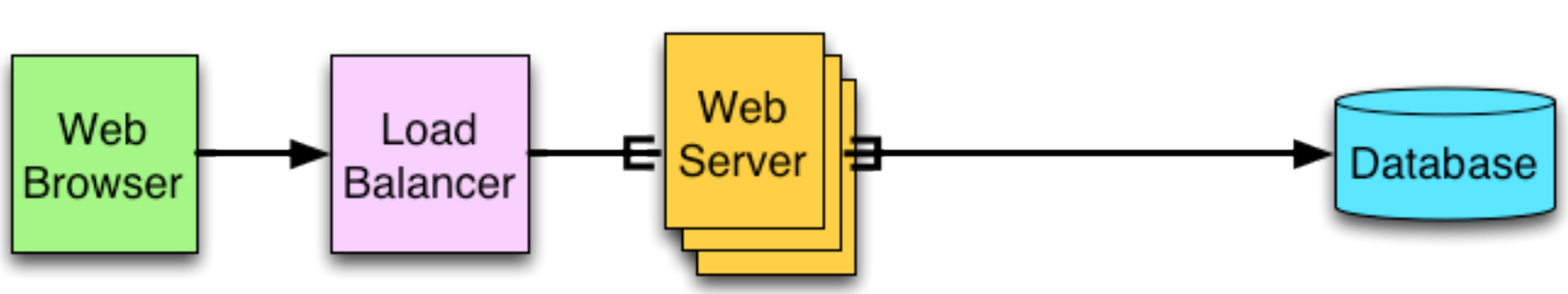
1 Tier (AKA Client/Server)



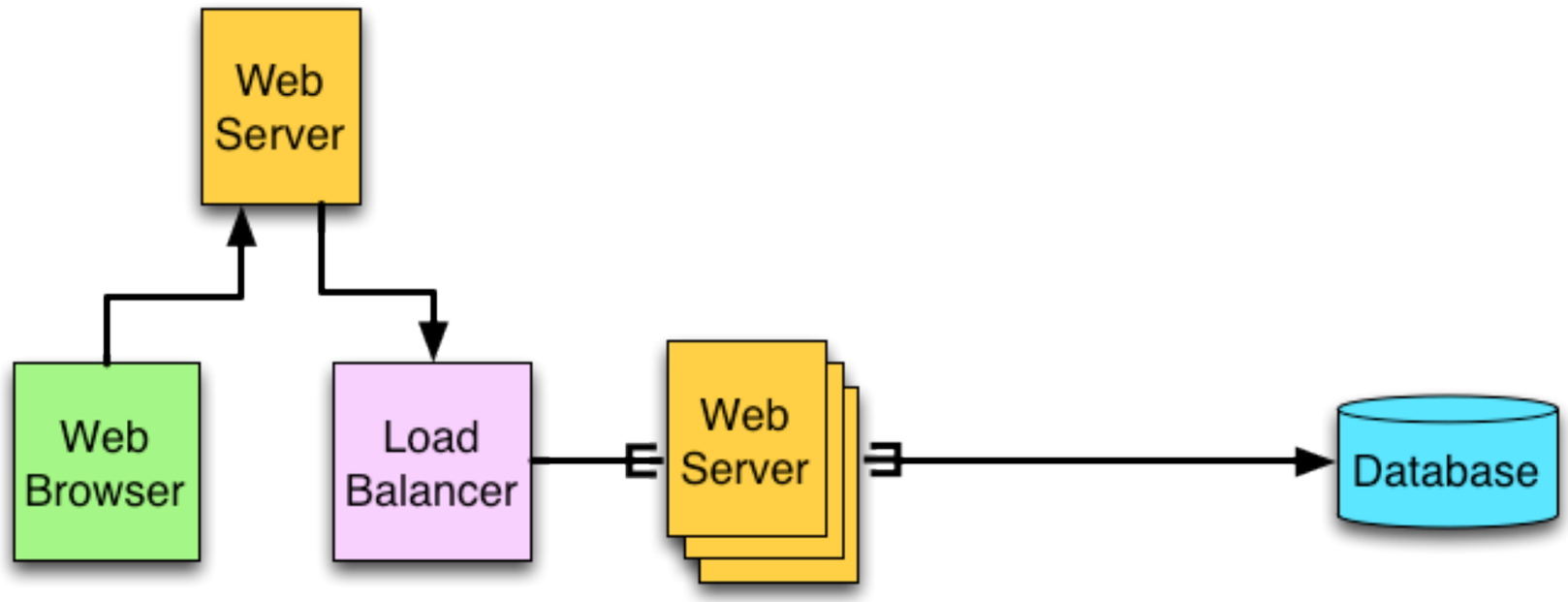
2 Tier



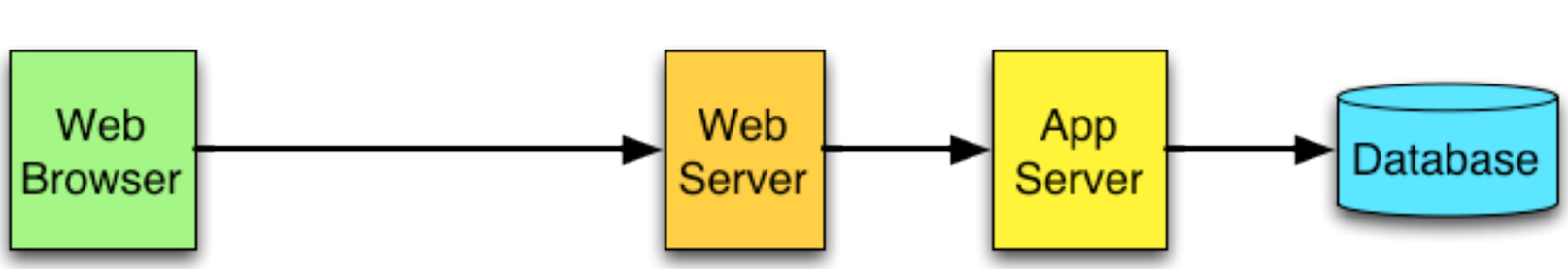
2 Tier Clustered



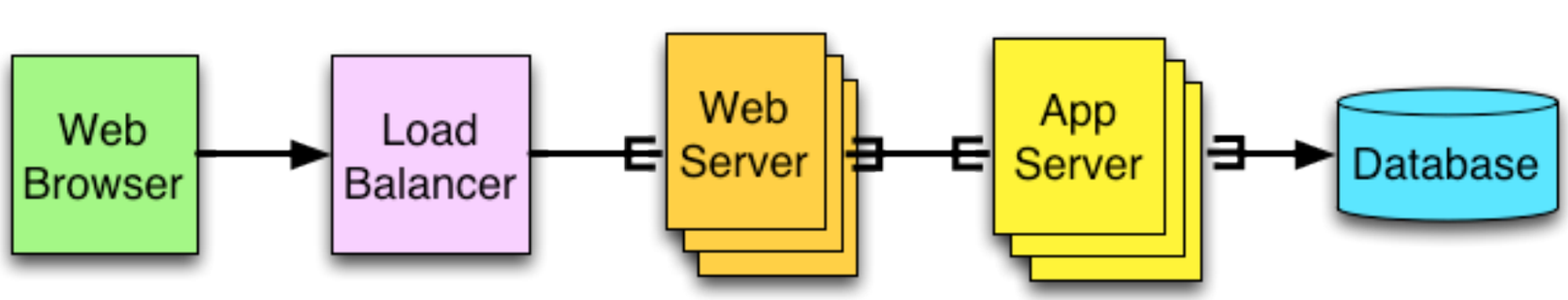
2 Tier - SAAS



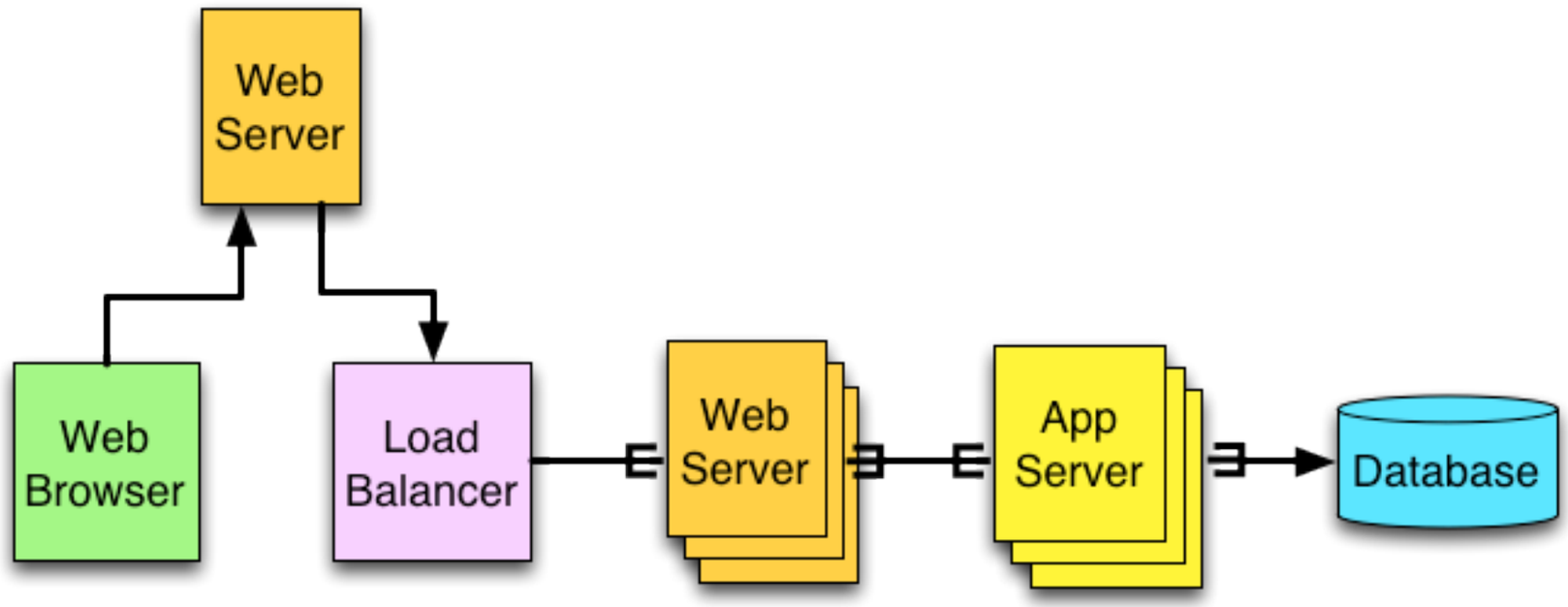
3 Tier



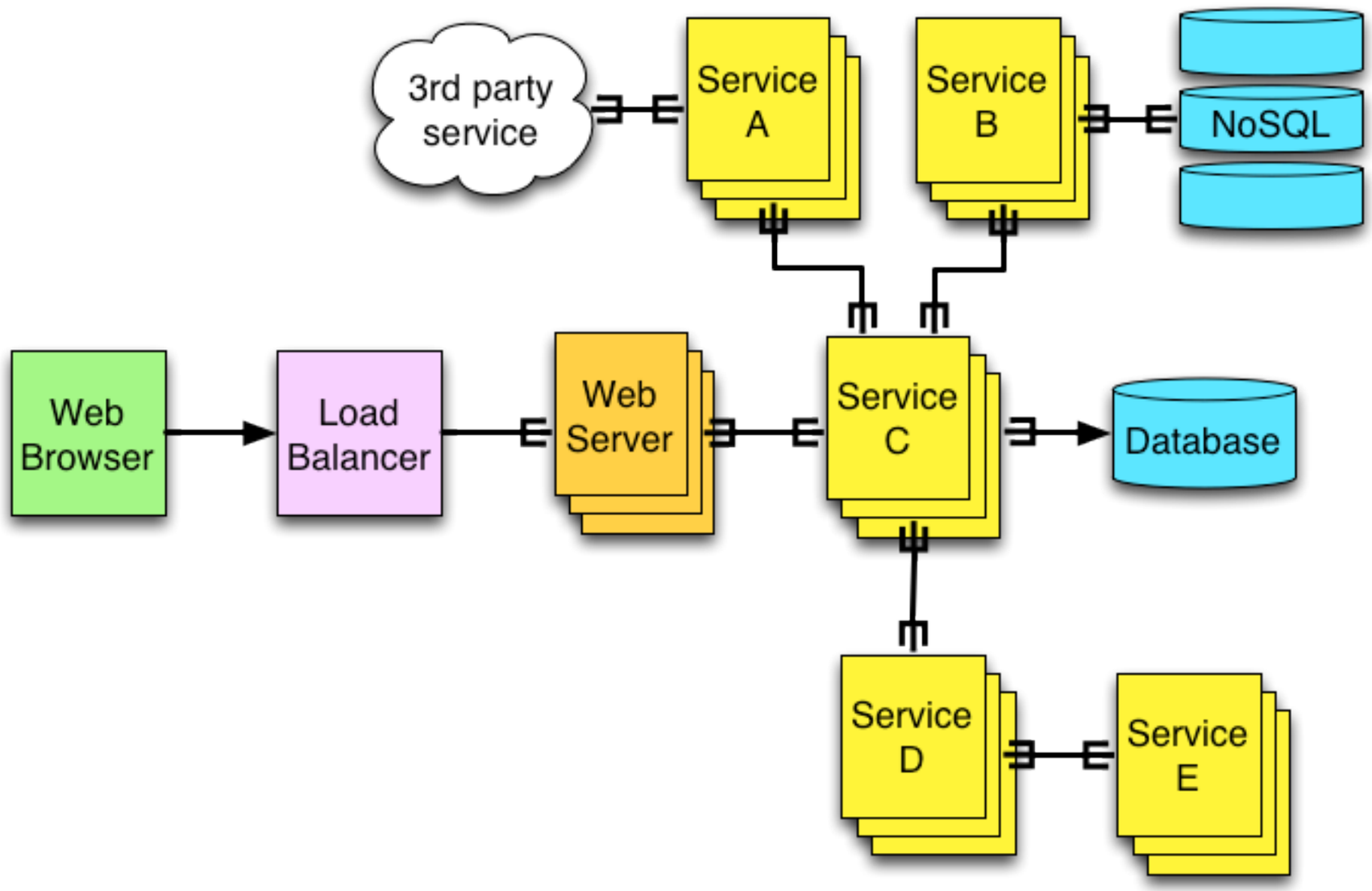
3 Tier Clustered



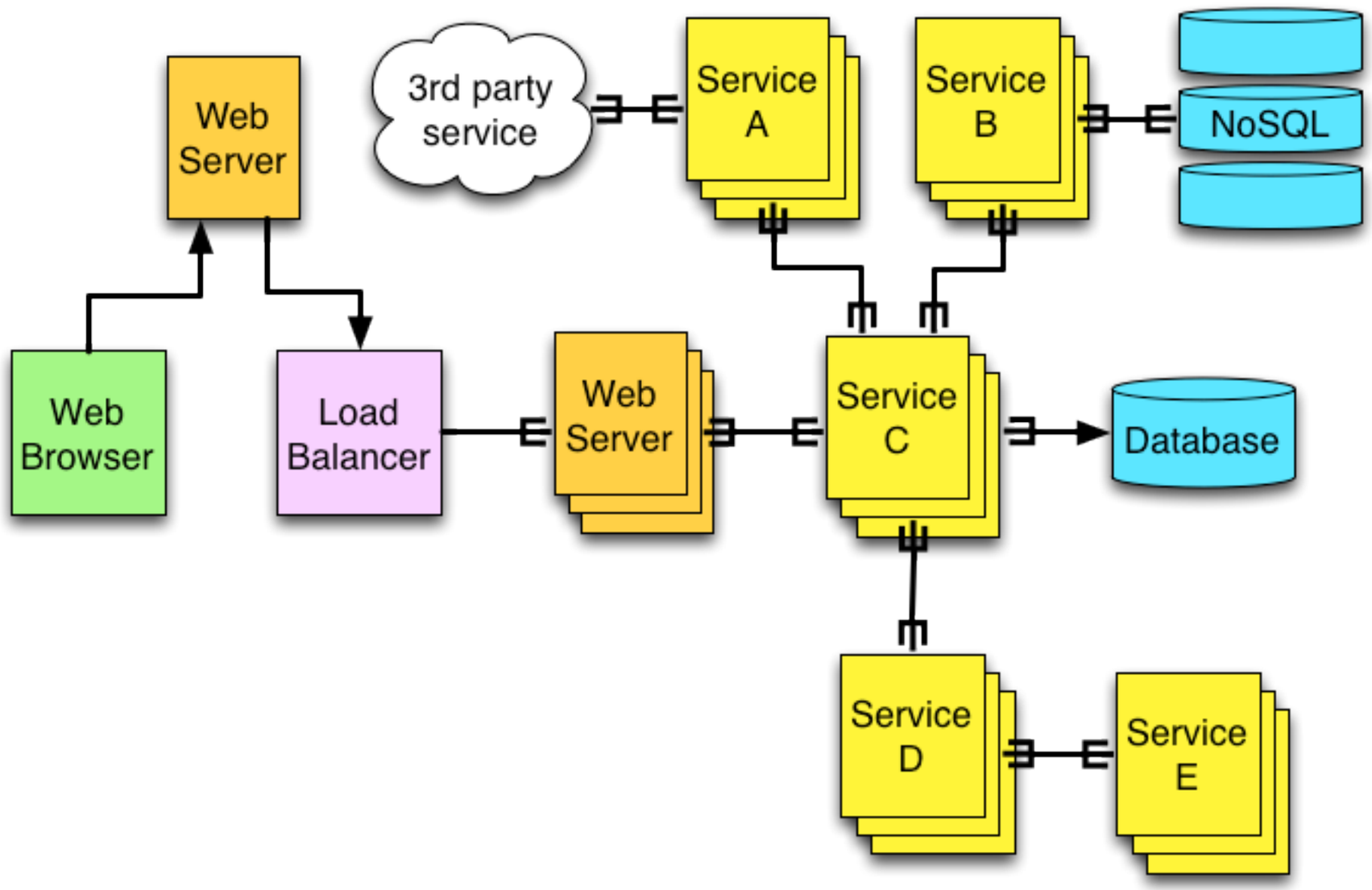
3 Tier - SAAS


Web
Server
Web
Browser
Load
Balancer
Web
Server
App
Server
Database

N Tier/SOA


3rd party service
Service A
Service B
NoSQL
Web Browser
Load Balancer
Web Server
Service C
Database
Service D
Service E

N Tier/SOA - SAAS



Troubleshooting for most

- Looking at logs for errors
- Capturing and viewing performance metrics, looking for visual patterns
- Try to reproduce errors based on a vague ticket description or a log line
- This becomes harder when you have dozens to hundreds of different systems to look through

Troubleshooting Distributed Systems

- Perform an internet search for "Troubleshooting Distributed Systems"
- "Traditional Approach"
  - Geared towards overall system performance monitoring
  - use NTP/synced ids/log everything you can/do something smart with it

The Trouble with the "Traditional Approach"

- Large data volumes (can be mitigated with sampling)
- Overhead on all requests (can be mitigated by going low level and forking packets)
- Geared towards general system performance and not application specific issues

What are Application Issues?

- A developer is trying to debug a web request bridging multiple subsystems
- A customer calls a support number and describes an issue
- A QA engineer is seeing unexpected results with a new feature or bug fix
- A sales engineer notices as issue while demoing a product

What could cause those issues?

- Actual bugs
- Data discrepancies
- Partially failed components or services
- PEBKAC

A Possible Solution

- Cross language tracing of requests
  - trigger a trace with an external input
    - log lots of extra stuff for that request to a central location via UDP
    - provide a way to view the data and drill down to the unexpected part

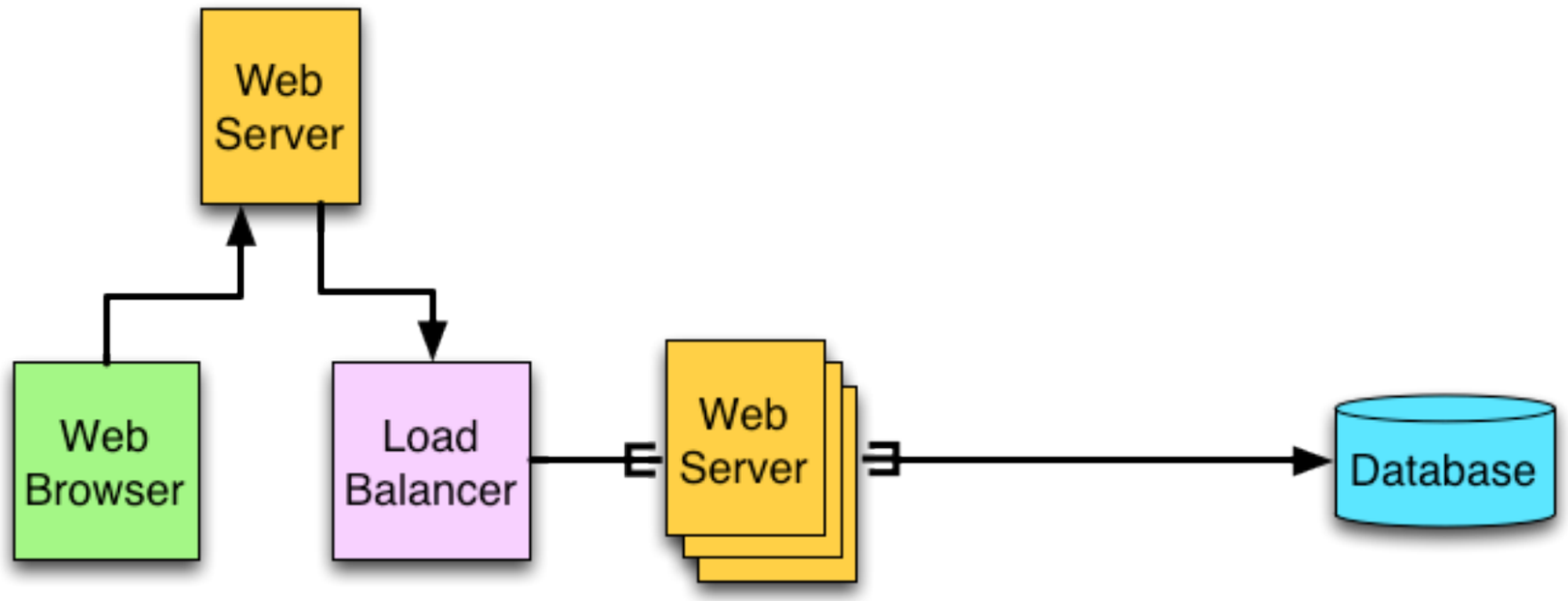
Evolution of a Solution

- 3 use cases
  - 2000: Search Advertising (Goto/ Overture)
  - 2004: Content Match (Yahoo)
  - 2010: Display Advertising (OpenX)

Search Advertising

- Given some keywords sent to a search engine
- Pick some Ads
- Include those ads in front of algorithmic results.
- Goto.com pioneered this in the late 90s
- Overture turned this into a service around 2000

2 Tier Distributed



Use Case

- Customer account manager gets a call from a customer who asks "Why am I not getting ads?"

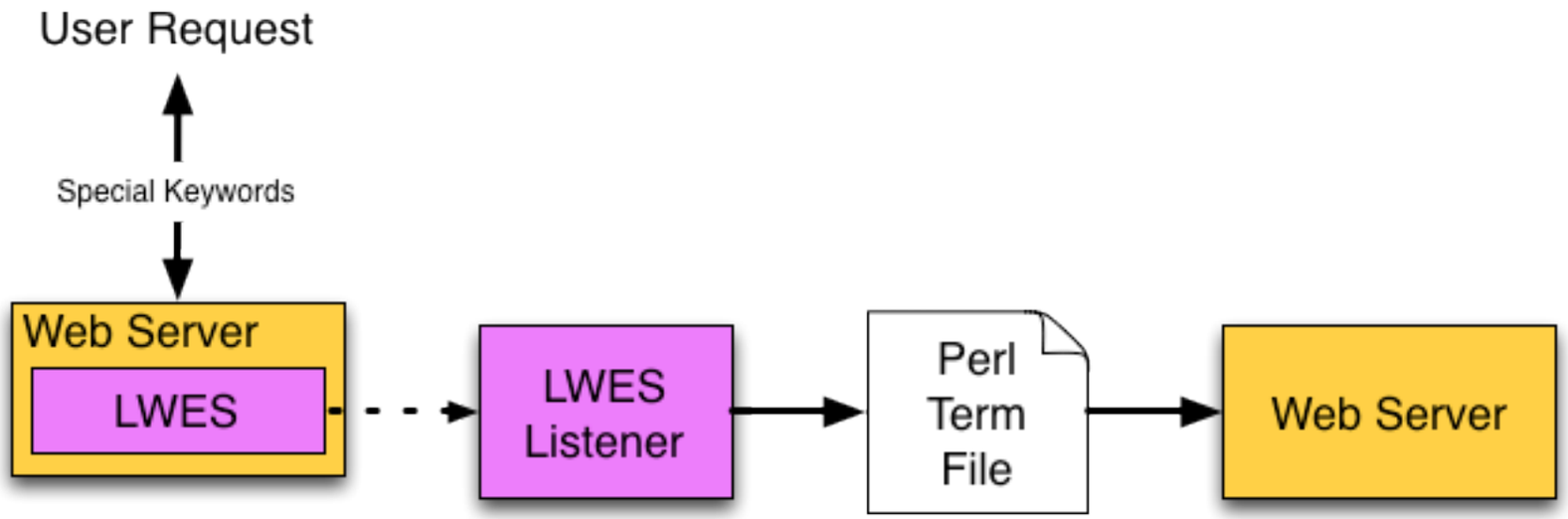
Tools Available

- Light Weight Event System - lwes
  - http://www.lwes.org/
  - cross language event system
  - UDP
  - fire and forget messages (low overhead on clients)

Solution

- isotope
  - demarcate a request via secret keywords
  - identify the request via the timestamp
  - if an isotope request, send lwes events containing perl data structures to a centralized server
  - dump to a file and serve up in a browser

Isotope


User Request
Special Keywords
Web Server
LWES
LWES Listener
Perl Term File
Web Server

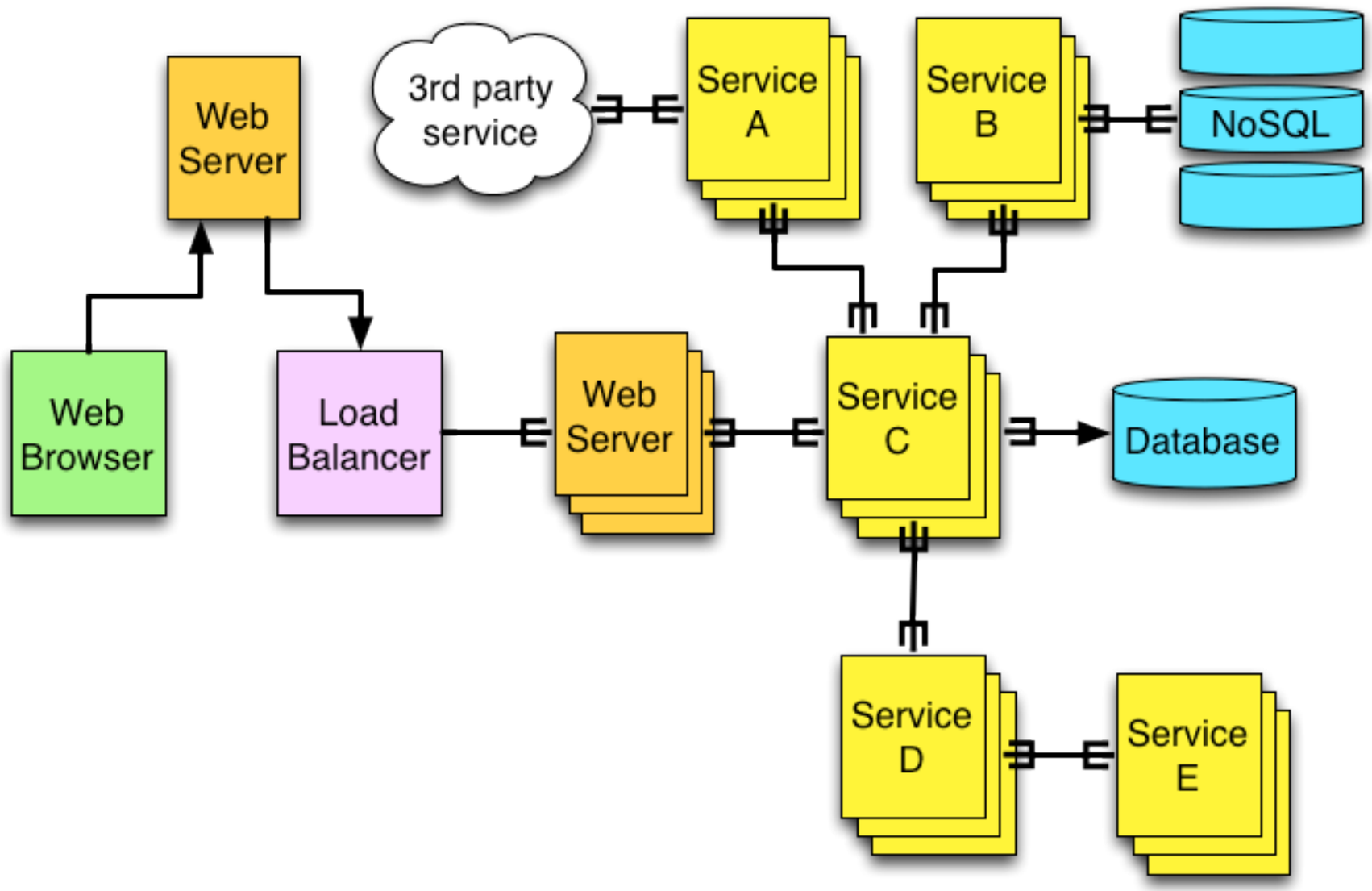
Lessons Learned

- timestamps can lead to conflicts, so need to add some other sort of id
- structured data can be useful
- making data accessible via an internal web service can be useful

Content Match

- Given the content of a web page
- Determine the subject
- Pick ads relevant to the subject
- Built this at Yahoo in the mid-2000s

N Tier


Web Server
3rd party service
Service A
Service B
NoSQL
Web Browser
Load Balancer
Web Server
Service C
Database
Service D
Service E

Use Cases

- Developers wonder "Where did my request go?"
  - which machines did it hit
  - what data did it use to make it's decision
- Customer support gets asked "Why am I not getting ads?"

Tools

- lwes again
  - multicast UDP
- command line listener of events
  - similar to lwes-event-printing-listener in lwes C distribution
  - able to filter based on an id

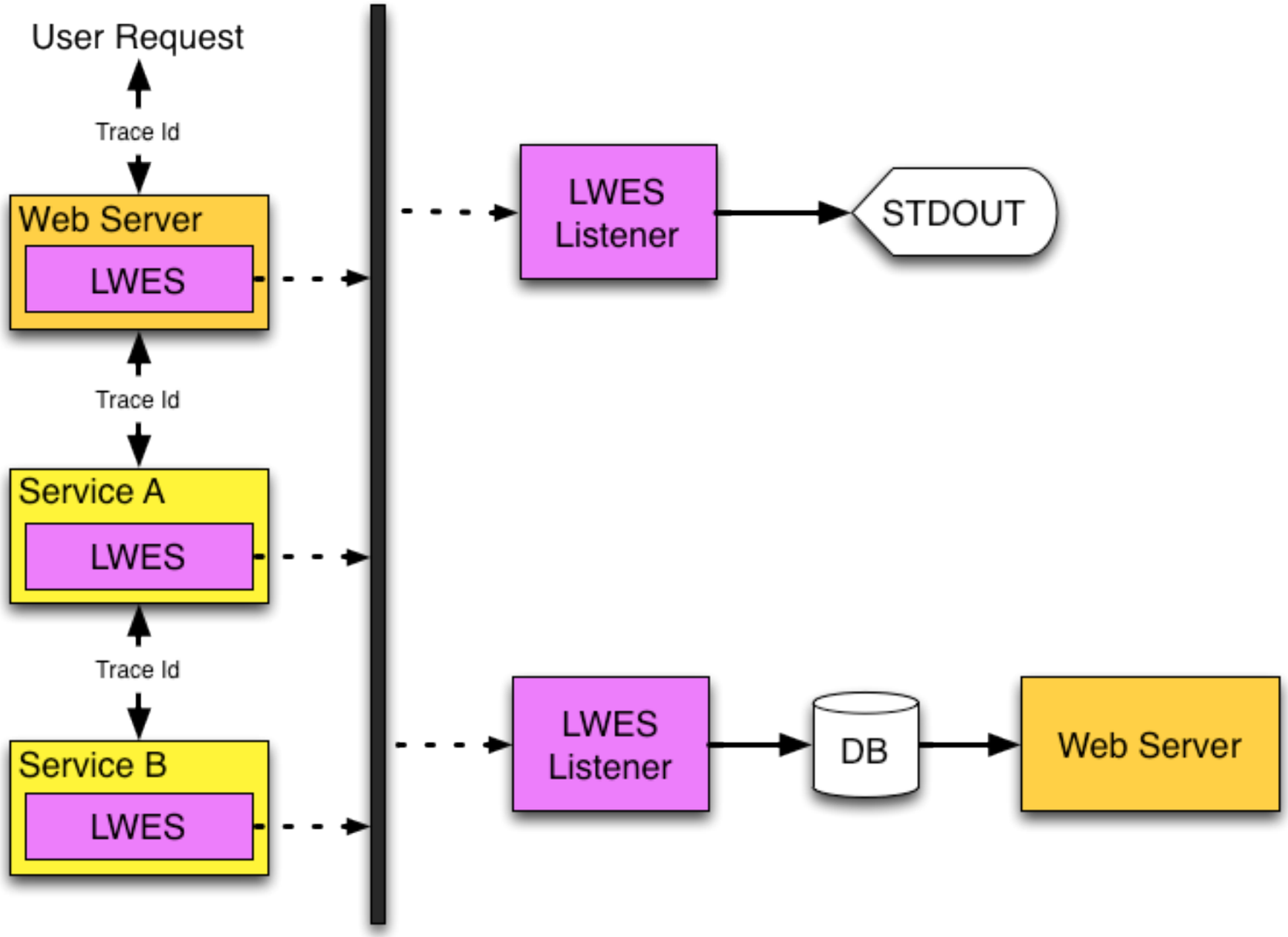
Solution

- llog
  - demarcate request with a secret query arg which accepted a non-zero positive integer
  - id was passed through all communications between components
  - when id is non-zero send extra information via multicast lwes to network
  - view trace in terminal

But what about customer support?

- Customer support couldn't use the command line tool
  - traces turned on for some number of requests
  - captured via multicast lwes and put into database
  - reports are generated

Trace


User Request
Trace Id
Web Server
LWES
Trace Id
Service A
LWES
Trace Id
Service B
LWES
LWES Listener
STDOUT
LWES Listener
DB
Web Server

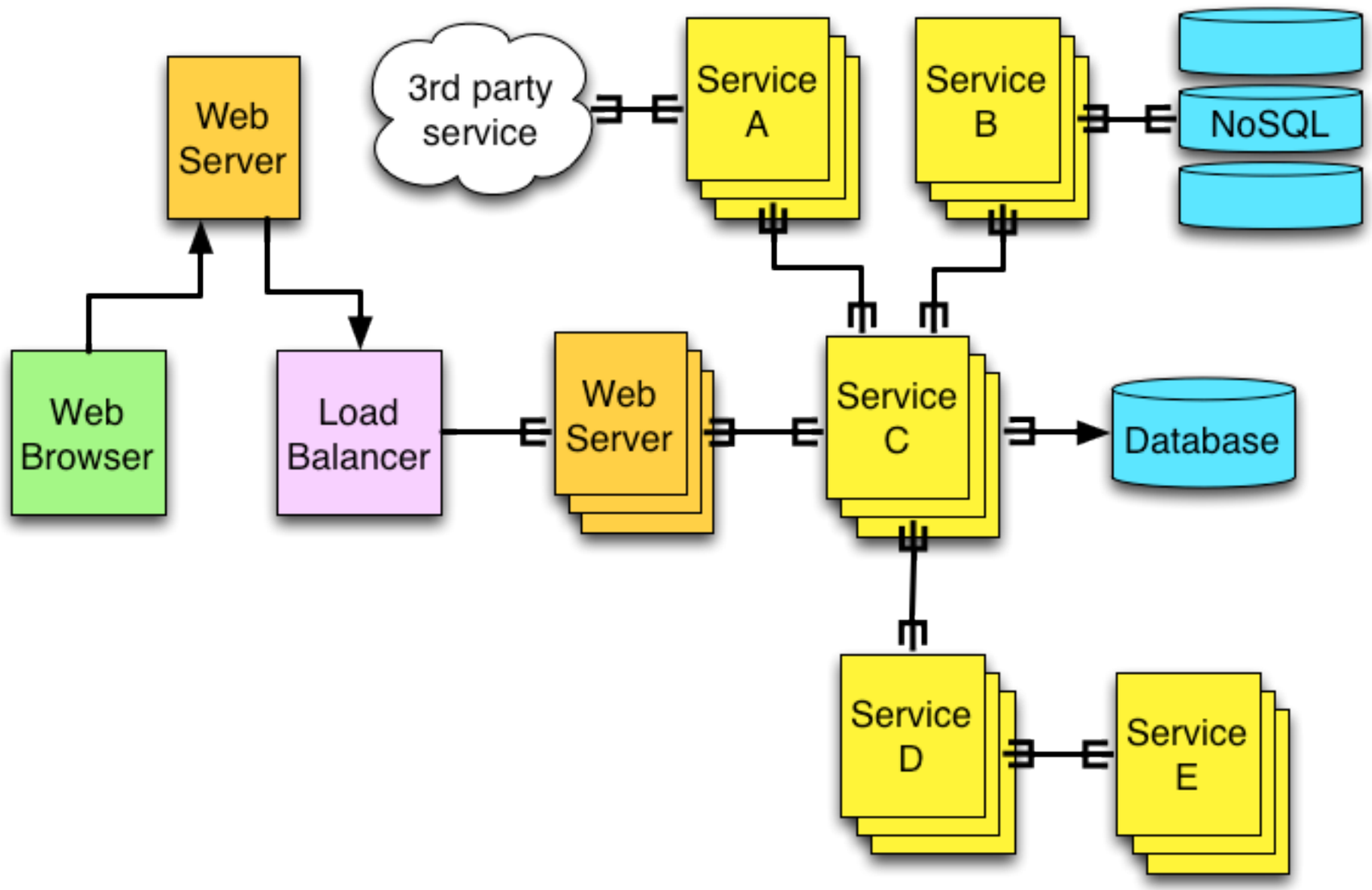
Lessons Learned

- Real time listening was useful for debugging, but there were many hacked together scripts to process trace information, and the output was not standardized so hard to parse
- Keeping around traces in a database for some time was very useful, but a relational database was limiting

Display Advertising

- Given a location on a webpage
- Pick the best ad for the user and webpage
- Currently doing this with OpenX

N Tier FTW!


Web Server
3rd party service
Service A
Service B
NoSQL
Web Browser
Load Balancer
Web Server
Service C
Database
Service D
Service E

Use Cases

- Why is my ad not showing?
- Where did my request go?
- How do I test a change to a subsystem?
- How do I find replication issues?

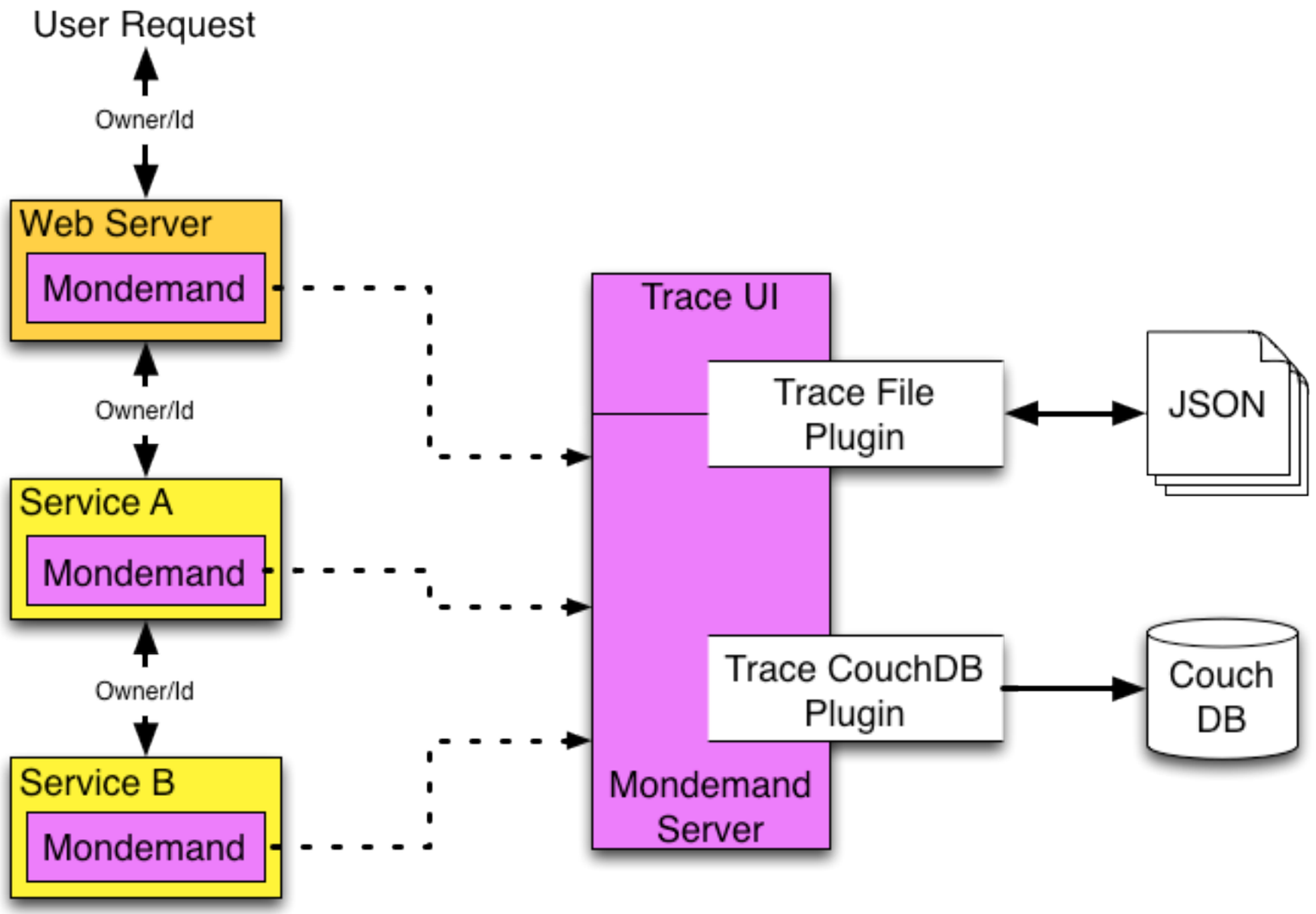
Tools

- lwes
- mondemand (http://www.mondemand.org/)
  - added structured output of stats/logs/traces on top of lwes
- mondemand-server
  - collects traces as JSON objects
  - simple UI for viewing

Solution

- demarcate request with a cookie containing two ids, an owner id and a trace id
- pass ids through to all services
- send trace messages to centralized server
- server captures and stores messages and provides UI for viewing

Mondemand


User Request
Owner/Id
Web Server
Mondemand
Owner/Id
Service A
Mondemand
Owner/Id
Service B
Mondemand
Trace UI
Trace File Plugin
JSON
Trace CouchDB Plugin
Couch DB
Mondemand Server

Lessons Learned

- A single id is not enough, you need at least 2 and possibly more
- The tool is useful for everyone from developers to QA to customer support
- Capture as much state as possible when tracing, you'll need it someday

Basic Examples

Erlang

```
mondemand:send_trace (
        webserver,              % identify program sending trace
        "trace_owner",          % owner of trace
        "trace_id",             % id for trace
        "received request",     % message
        [])                     % extra data
```

Java

```
// identify program sending trace
client = new Client ("webserver");

HashMap<String, String> tmp =
  new HashMap<String, String> ();

client.traceMessage (
    "trace_owner",          // owner of trace
    "trace_id",             // id for trace
    "received request",     // message
    tmp);                   // extra data
```

Command Line

```
mondemand-tool -o lwes::127.0.0.1:20502 \
               # identify program sending trace \
               -p webserver \
               # Owner of trace : id for trace : message \
               -T "trace_owner:trace_id:received request"
```

Mondemand JSON


```
{
   "SenderIP": "127.0.0.1",
   "SenderPort": 52823,
   "ReceiptTime": 1392874916206,
   "EventName": "MonDemand::TraceMsg",
   "mondemand.src_host": "renym.local",
   "mondemand.prog_id": "webserver",
   "mondemand.owner": "trace_owner",
   "mondemand.trace_id": "trace_id",
   "mondemand.message": "received request"
}
```

Examples with Embedded JSON

Erlang

```
mondemand:send_trace (
        webserver,                  % identify program sending trace
        "trace_owner",              % owner of trace
        "trace_id",                 % id for trace
      "received request", % message
      [ { extra,                          % extra data can contain
          "{\"key\":\"value\"}"           % json strings
        }
      ])
```

Java

```
// identify program sending trace
client = new Client ("webserver");

HashMap<String, String> tmp =
   new HashMap<String, String> ();
tmp.put ("extra",                    // extra data can contain
         "{\"key\":\"value\"}"   // json strings
        );

client.traceMessage (
     "trace_owner",              // owner of trace
     "trace_id",                 // id for trace
     "received request",         // message
     tmp);                       // extra data
```

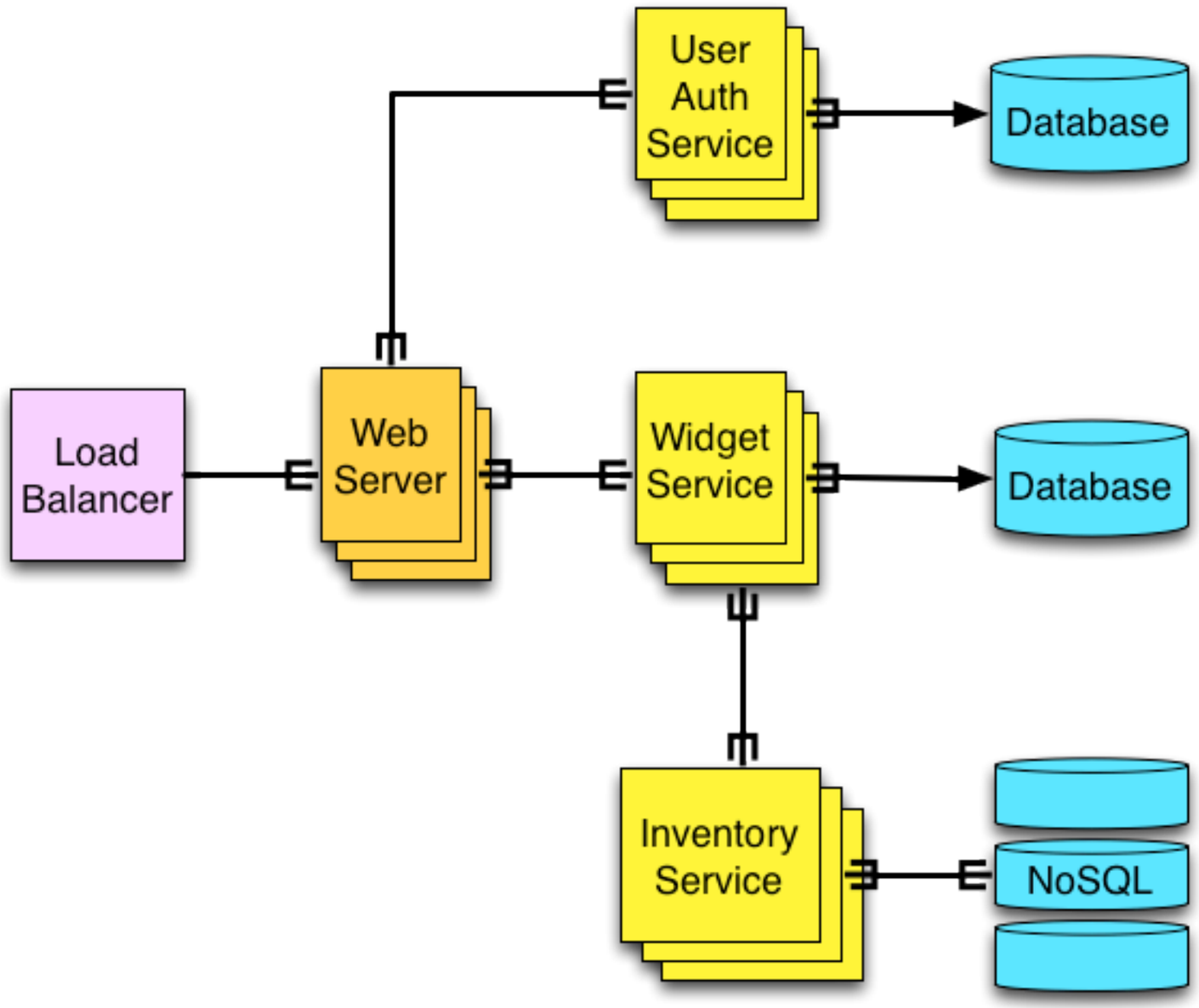
Command Line

```
mondemand-tool -o lwes::127.0.0.1:20502 \
               # identify program sending trace \
               -p webserver \
               # Owner of trace : id for trace : message \
               -T "trace_owner:trace_id:received request" \
               # extra data can contain json strings
               -t "extra:{\"key\":\"value\"}"
```

Mondemand JSON


```
{
  "SenderIP": "127.0.0.1",
  "SenderPort": 64613,
  "ReceiptTime": 1392875074968,
  "EventName": "MonDemand::TraceMsg",
  "mondemand.src_host": "renym.local",
  "mondemand.prog_id": "webserver",
  "mondemand.owner": "trace_owner",
  "mondemand.trace_id": "trace_id",
  "mondemand.message": "received request",
  "extra": { "key": "value" }
}
```

Demo of UI


User
Auth
Service
Database
Load
Balancer
Web
Server
Widget
Service
Database
Inventory
Service
NoSQL

Final Thoughts

- When building new systems
  - add the ability to add ids to a request in some ad hoc manner
  - pass the ids throughout the system
  - this lays the foundation for any number of tracing setups

Limitations/ Future Work

- Large objects in traces
  - UDP packet limits trace sizes
- QueAsy system for feeding traces back into a system as test cases

Questions?

Thanks!

- http://www.lwes.org/
- http://www.mondemand.org/
- http://github.com/djnym
- anthony.molinaro@openx.com
- anthonym@alumni.caltech.edu