# Orchestrating Your Cloud-orchestra

Abram Hindle
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
abram.hindle@ualberta.ca

## ABSTRACT

Cloud computing potentially ushers in a new era of computer music performance with exceptionally large computer music instruments consisting of 10s to 100s of virtual machines which we propose to call a 'cloud-orchestra'. Cloud computing allows for the rapid provisioning of resources, but to deploy such a complicated and interconnected network of software synthesizers in the cloud requires a lot of manual work, system administration knowledge, and developer/operator skills. This is a barrier to computer musicians whose goal is to produce and perform music, and not to administer 100s of computers. This work discusses the issues facing cloud-orchestra deployment and offers an abstract solution and a concrete implementation. The abstract solution is to generate cloud-orchestra deployment plans by allowing computer musicians to model their network of synthesizers and to describe their resources. A model optimizer will compute near-optimal deployment plans to synchronize, deploy, and orchestrate the start-up of a complex network of synthesizers deployed to many computers. This model driven development approach frees computer musicians from much of the hassle of deployment and allocation. Computer musicians can focus on the configuration of musical components and leave the resource allocation up to the modelling software to optimize.

## Author Keywords

cloud computing, cloud instruments, cloud-orchestra

## 1. INTRODUCTION

With the flick of a switch your lights turn on. With the click of the mouse your computation can turn on too. Cloud computing promises utility computing: computing treated as a utility and billed like a utility – even provisioned like a utility such as electricity [5]. This demand-based dynamic provisioning of virtual machines is what makes it possible for cloud providers to allow clients to scale computation to their needs. Cloud computing promises much flexibility but these promises are often wrapped in intense complexity [5].

The benefits of using cloud computing are clear: one does not need to own the hardware to run their software; cloud computing enables experimentation with large allocations of resources for short periods of time; cloud service providers often have reasonable bandwidth; performers who exploit cloud services avoid bringing lots of computer equipment to a performance venue [10]; networked computers enabled centralized and decentralized collaborative instruments [12].

The cloud has been leveraged for computer music performance [10]. While the authors succeed at a demoing the feasibility of cloud instruments, they also demonstrated the complexity and frustration of organizing and orchestrating synthesizers in the cloud. Organizing these synthesizers requires much knowledge about cloud computers, networking, systems administration, Unix-like systems, shell scripting and programming. Essentially such a prerequisite knowledge is a large barrier to the adoption of cloud computing for computer music performance.

Requiring a computer musician to play the role of system administrators or developer/operators is too high a barrier. This work seeks to enable the generation and deployment of a *cloud-orchestra*. We define a cloud-orchestra as a network of software synthesizers deployed to multiple networked computers (usually on a cloud). We envisage cloud-orchestras to be composed of multiple virtual machines running software synthesizers connected together by network audio links.

Deployment is an ardous task that requires much networking, programming, and shell scripting knowledge – unless it is automated. Systems can be built that take both a model of a cloud-orchestra and a list of resources as input to produce a coherent, runnable deployment plan. Once the deployment plan is generated the computer musician can deploy/start a cloud-orchestra upon request.

*Model Driven Development* (MDD) [9] allows for the rapid modelling (defining) and generation of runnable code that implements such a model. We propose to abstractly model a cloud-orchestra and rely upon model optimizers to fit such a synthesizer network near-optimally to available resources such as cores, virtual machines, etc.

This work addresses: the difficulty of configuring and deploying a cloud-orchestra; and the efficient allocation of resources for cloud-orchestras. These problems motivate the following contributions described in this paper:

- Propose and define cloud-orchestras;
- Define a model of cloud-orchestras;
- Argue for a declarative model driven development approach to cloud-orchestras;
- Provide a prototype that defines, generates, and deploys a cloud-orchestra;
- Address cloud-orchestra deployment abstractly and concretely.

## 2. PREVIOUS WORK

Cloud-orchestras can employ model driven development, cloud computing, cloud deployment, networked audio, net-


*NIME'15,* May 31-June 3, 2015, Louisiana State Univ., Baton Rouge, LA.

worked computer music, networking latency, streaming technology and networked orchestras.

*Model Driven Development* (MDD) and *Model Driven Engineering* [9] are methods of specifying software and systems as models and then relying on model checkers, and model generators to generate skeletal stub code, or fully working systems. Model Driven Development is often used in fields which have computation but are limited to certain problem domains, often allowing end-users to customize a system without much programming experience.

Barbosa et al. [3] surveyed networked computer music since the 1970s. This body of work lead to the `jack` [7] low latency sound server, netjack [7] and Jacktrip [6]. The Netjack and Jacktrip [6] send synchronized `jack` audio streams over a LANs and the internet.

One reason to employ cloud computing in a computer music instrument is to provide user interfaces to mobile devices over the web, thereby allowing audience participation. Oh et al. [13] demonstrate the use of smartphones in audience participatory music and performance. Jordà [11] describes many patterns of collective control and multi-user instruments as well as the management of musicality of instruments. Cloud computing was indirectly used by Dahl et al. [8] in TweetDreams. TweetDreams allows an audience to participate with a musical instrument by tweeting at a twitter account that aggregates audience tweets using Twitter's own message delivery cloud. These recommendations would be valuable for anyone creating an audience-participatory instrument with a cloud-orchestra.

Jesse Allison et al. [1, 2] describe the Nexus framework to allow for user-interface distribution via the web. While Weitzner et al. [17] built *massMobile* that allows the creation of a web interface to interact with Max/MSP via the web. Both works emphasize the value of HTML5 interfaces as they are both standardized and ubiquitous. Exposing these user-interfaces from a cloud-orchestra would prove invaluable as it would allow fine grained control of a synthesizer running on a VM.

Cloud-orchestras share many of the same problems and difficulties as laptop orchestras [15, 14]. Laptop orchestras are composed of laptop synthesizer users who are connected together over OpenSound Control or `jack`. The main difference between a laptop orchestra and a cloud-orchestra is that a cloud-orchestra doesn't necessarily have more than 1 performer or musician, much of the orchestra is under computer control.

Lee et al. [12] describe many opportunities for live network coding. They argue that networked live coded music allows for interesting mixes of centralized and decentralized synthesis and control – all amenable to cloud-orchestras.

Ansible, Chef and Puppet [16] are common configuration management and automation framework tools. These tools tend to be used to install software, synchronize software, bring up servers and automate tasks. In this research we use Ansible to run commands on multiple VMs.

Beck et al. [4] first used grid-computing and generated deployment plans for interactive performances. Hindle [10] describes using the cloud [5] for computer music and the issues one encounters, exporting audio from the cloud. Cloud-orchestras suffer from this problem, in the cloud no one can hear unless you stream. This work seeks to elaborate on the prior work and focuses more on synthesizers in the cloud.

# 3. MODEL OF A SYNTH NETWORK

The proposed solution to the cloud-orchestra deployment problem is to leverage model driven development to define a cloud-orchestra. A definition of a cloud-orchestra is a model. This model may be optimized by model optimizers which search for an efficient configuration. Once such a configuration has been found, templates can be executed to generate a working deployment plan. This deployment plan can be executed and the cloud-orchestra will be instantiated, configured, and executed eventually producing music in near real-time streamed over a network.

Cloud-orchestras and their potential resources can be modelled and these models can be used to develop a deployment plan to deploy a cloud-orchestra on resources available to you. The fundamental problem of a declarative model of a cloud-orchestra is how to match a set of resources (hosts) with the workload (cloud-orchestra) proposed.

The inputs to this problem are a set of usable host virtual machines and a synthesizer network model (the cloud-orchestra). The model we use is partially described in the *model* package of Figure 2. Synthesizers are modelled as *SynthModules* or *modules*. An instance of a module is a *SynthBlock*. The distinction is similar to the distinction between a class and an object. The SynthBlock has a SynthModule, a name and a host will be associated with it. These Synthblocks are collected within a *SynthDef*, which is the definition of the entire cloud-orchestra. The SynthDef also contains connections that join the inputs and outputs of a SynthBlock together. In the next section we formalize this model for the purposes of optimization.

## 3.1 Modeling and optimization

In general one can model the resource allocation problem as a cost minimization problem.One can formalize this problem as an Integer Linear Programming problem, and potentially minimize it using tools such as GNU Linear Programming Kit [1]. This formulation enables search methods to produce optimal or near-optimal configurations.

Regardless, given available resources or a resource allocation one must allocate synthesizers (synthBlocks) to hosts. For a cloud-orchestra deployment one should consider optimizing for:

- Locality: the more synths communicating on the same VM the less latency and network bandwidth required.
- CPU load: too many synths will overload a CPU hamper near-realtime performance.
- Remote Links: The number of remote links between synths should be reduced, different allocations of synths to different hosts can increase or decrease the number of remote links.
- Bandwidth: network bandwidth use decreases as locality of synths increases and the number of remote links decreases.

Other costs could be considered such as the monetary cost to provision cores, VMs, memory, and bandwidth. One could add another level of modeling to model provisioning to determine the minimum resources required to deploy a cloud-orchestra.

The optimizer's main task is to take the available hosts and allocate the synthesizer instances (synthBlocks) to these hosts based on the cores and bandwidth available. The allocation should be optimal in the sense that it increases locality and reduces the number of remote links. Reducing remote links limits the potential latency.

Given synthBlocks from $S$, connections can be represented as a set of edges $C_{s,t}$ from $C : S \times S$ where $s$ and $t$ are synthBlocks and a connection from $s$ to $t$ is represented by set membership. The UML in Figure 2 is labelled with our mathematical definitions.

[1] `https://www.gnu.org/software/glpk/`

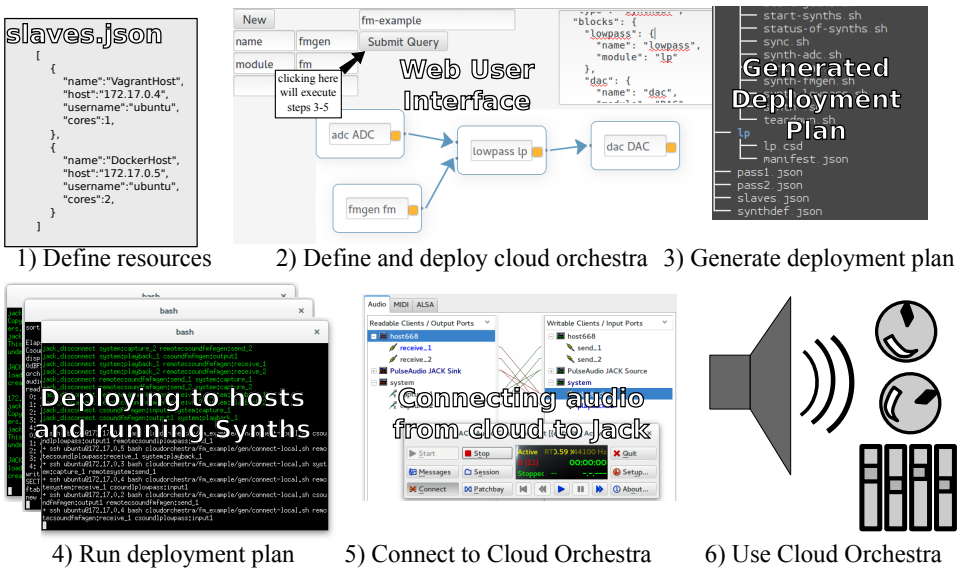


**Figure 1: Example of defining and deploying a cloud-orchestra using the concrete implementation *synth-cloud-orchestra*.**

This synthDef model tuple $(H, S, C, A)$ can formalized and optimized using the following definitions and mappings:

- $H$ is the set of hosts.
- $S$ is the set of SynthBlocks.
- $C : S \times S$ is the set of connections between SynthBlocks. $C_{s,t}$ where $s$ and $t$ are synthBlocks represents a connection between $s$ and $t$. This set is constant throughout optimization.
- $A : H \times S$ is the allocation of synthBlocks to hosts where $A_{h,s}$, $h \in H$ and $s \in S$, represents an allocation of a synth $s$ to a host $h$. This set is the set to be optimized.
- $cores : H \mapsto \mathbb{Z}^+$ is a function returning the number of cores of a host.
- $blocks : H \mapsto \mathbb{Z} = |\{A_{h,s} | s \in S\}|$ returns the number of synthblocks allocated to the host in $A$.
- $overflow : H \mapsto \mathbb{Z} = max(0, blocks(h) - cores(h))$ The number of allocated synthBlocks beyond the cores of the host.
- $overflow_{all} = \sum_{h \in H} overflow(h)$ The number of overflowing hosts for the whole network.
- $connections : H \mapsto \mathbb{Z} = |C_{u,x}| + |C_{x,u}|$ where $x \in S$ and $u \in \{s | s \in S \wedge A_{h,s}\}$. The number of connections to/from synthBlocks of host $h$.
- $locals : H \mapsto \mathbb{Z} = |\{C_{s,t} | A_{h,s} \wedge A_{h,t} \wedge C_{s,t}\}|$ The number of connections internal to host $h$.
- $remotes : H \mapsto \mathbb{Z} = connections(host) - locals(h)$ The number of remote connections to and from host $h$.
- $remotes_{all} = \sum_{h \in h} remotes(h)$ The sum of remote connections over all hosts.

An optimizer for this problem should be a function that at least takes in a current configuration $optimize : (H \times S \times C) \mapsto (A : H \times S)$ and produces a host to synthBlock mapping $A$. When implemented it could potentially take a partial initial allocation $A$ if needed.

The goal of the *optimize* function should be to minimize $overflow_{all}$ first and then to minimize $remotes_{all}$. A synthDef that needs more resources than are available is a concern. A tuning parameter $\alpha$ ($\alpha \in \mathbb{R}$) can be used to indicate the importance of $overflow_{all}$ versus $remotes_{all}$. An optimal optimizer function will meet the requirements of Equation 1 below:

$$optimize(H, S, C) = \arg\min_{A} f(A)$$
$$f : A \mapsto \mathbb{R} = \alpha \cdot overflow_{all} + remotes_{all} \quad (1)$$

Naively one could generate all permutations of the legal sets $A$, matching SynthBlocks and Hosts, and rank them by function $f$ described in Equation 1, choosing the minimum result. There are likely more efficient methods of determining optimal configurations, such as employing linear programming. At this point, once an optimal or near-optimal configuration of hosts and synthBlocks is found, the deployment can be generated or executed.

## 3.2 Deploying a synth network

In this section we describe a general process to start and deploy a cloud-orchestra. Any concrete implementation will follow this process. After generating all the code necessary to deploy and run a cloud-orchestra from the model, one has to deploy and run the synth network.

1. First the synthesizer source code and cloud-orchestra model are synchronized to all synth hosts (rsync).
2. Then the sound server (jackd) is started on each synth host.
3. Remote network audio connectors are started (jacktrip or netjack).
4. For every synth host, their synthBlocks' associated synthesizers are started.
5. All audio connections are disconnected on all hosts to avoid interference and to start in a disconnected state.
6. SynthBlock synthesizers are connected to their appropriate `jack` audio ports per each synth host.
7. An audio exporter is started (e.g., icecast (mp3), jacktrip, or cloudorch [10] (websocket streaming)).

To halt or tear down the cloud-orchestra each host is contacted and all sound servers, remote network audio connectors, and all synthesizers and audio exporters are killed.

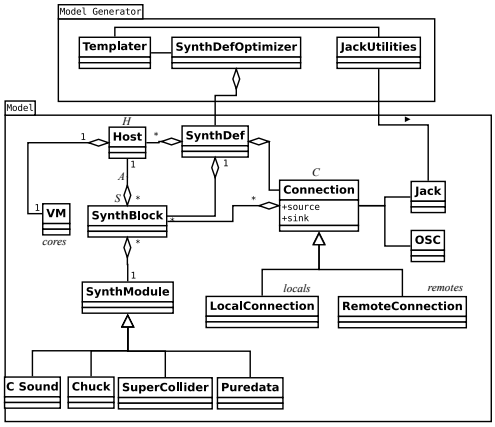


**Figure 2: Architecture of an example cloud-orchestra**

The responsibilities of a concrete framework based on this abstract will be: to elicit a definition from a user; to elicit resources and resource constraints; to search for optimal deployment plans; to generate or execute such a deployment plan; to enable the end-user to listen to the cloud-orchestra once it is deployed. These responsibilities will be discussed in the next section.

# 4. A CONCRETE IMPLEMENTATION

Our concrete cloud-orchestra generator is called *synth-cloud-orchestra* and it is freely available [2]. The hope is that the compute music community can take advantage of this operable framework and start building cloud-orchestras without the hassle that the authors went through to get the cloud-orchestra to run.

A dockerfile is provided with synth-cloud-orchestra and acts as instructions to setup hosts consistently so they can act as synth hosts. This implementation currently makes some assumptions:

- The cloud-orchestra user has write access to a home directory of the hosts they will employ.
- The hosts already have the main software synthesizers needed installed (Csound, Chuck, Supercollider, Pd).
- The hosts allow for passwordless login via ssh or ssh-agents.
- The hosts can access each other over ports used by `jack` and jacktrip.
- The hosts need bash and ruby installed.
- The head node needs ansible, bash, and ruby installed.

The current implementation generates a series a bash scripts that allows for the synchronization of files, deployment, and the startup and connecting of synthesizers. Once generated all an end-user needs to run is `gen/run.sh` which will execute all of the steps described in Section 3.2 on deployment.

Throughout this section we will use an example of a FM synthesizer and microphone (ADC) chained through a low-pass filter to a DAC. While these are not expensive operations they can be run on separate machines. Figure 1 (b) shows a graphical configuration the full webUI and shows some of the JSON definition of the synthdef.

[2]Demo and source code: `https://archive.org/details/CloudOrchestraDemo` `http://github.com/abramhindle/synth-cloud-orchestra`

*Defining resources:.*

A `slaves.json` (an example is in Figure 1) lists all machines potentially under the cloud-orchestra's control. Each definition is named and also details the ssh username used for access, the hostnames, and the number of cores.

## 4.1 Defining the cloud-orchestra

By seperating the cloud-orchestra definitions from the resources, the model optimizer is enabled to come up with many solutions and choose the best ones based on the resources available and the optimization heuristics which emphasize locality preserving configurations. Figure 1 shows how the webUI represents and generates a JSON definition. The user can define cloud-orchestras in either pure JSON or with the webUI that comes with synth-cloud-orchestra.

Figure 1 depicts part of the web-based user interface provided by synth-cloud-orchestra. The boxes are instances of synthesizers, synthBlocks, and the arrows are the directional links between the synthBlocks. By dragging from the boxes on the synthBlocks to the other synthBlocks themselves one can connect the synthBlocks together. The synthBlocks are specified by first specifying a name of that instance of the synthesizer and then the kind of synthesizer it is. Double clicking deletes connections and synthesizers. New synthBlocks are added by clicking the `new` button.

This method is flexible and allows optional parameters to be specified. For instance the `host` parameter in the SynthBlocks can be specified, this allows the user to override the optimizer and assign a particular synthblock to a particular host, perhaps for performance reasons. Hosts that do not reference particular slave hosts will be automatically allocated from the `slaves.json` file.

Synth-cloud-orchestra comes with some default synthesizers including ADC and DAC, these are meant to represent general inputs and outputs that might be in the cloud or actual computers. Audio maybe exported from this cloud-orchestra using jacktrip, but cloudorch [10] – the soundcard in the cloud – or icecast can also be used. Currently synchronization is the responsibility to the audio transport: `jack` and jacktrip. Control signals can use OSC but are not synchronized.

## 4.2 Adding synthesizers

Each custom synthesizer is considered a *synthModule* or *module*. An optional manifest file, `manifest.json` defines module's name and the synthesizers source code. For instance the example `fm` synthesizer is a `csound` synthesizer with the `fm.csd` as the synthesizer to run and its module name is `fm`.

A user can add a custom synth to their cloud-orchestra by creating a directory that will serve as their module. This directory will contain all assets and code required by their synthesizer including Csound orchestras, SuperCollider source code, and sound assets. These modules will be synchronized to the appropriate hosts and executed as needed.

By using templates one can avoid hard-coding IP addresses in the synthesizer's definition one can treat the synthesizer source code as a template to parameterize. Thus at generation time, the appropriate configuration will be inserted into the source code of the synthesizer, as well as dynamicly into environment variables.

This pattern of user-defined synthesizers allows for the computer musician to rely on their own synthesizers and also leverage the hard work of the makers of CSound, SuperCollider, Chuck, Pd, and other software synthesizers.

## 4.3 Deploying a cloud-orchestra

The program `synthdefrunner` is run within the directory defining the cloud-orchestra. This program reads the definitions of the cloud-orchestra and available resources. It then optimizes the network according to heuristics meant to improve locality. If there are not enough cores for all of the synthesizers then it tries not overload the cores too much and tries to increase locality. The current implementation is naive and assumes at peak utilization 1 synthesizer needs at least 1 core, and cores are not shared. Regardless, the best suggested network configuration is chosen and recorded. Based on this configuration, the Templater that generates the `gen` directory from the `gen.erb` templates. These are shell scripts, written as very portable `bash` scripts that define how to run the synthesizers and how to connect the audio links together.

Once `gen` is created the user can simply run `gen/run.sh` to synchronize, deploy, instantiate and use their cloud-orchestra as per Section 3.2 on deployment. The WebUI lets a user define a cloud-orchestra synthdef and deploy it in one step, clicking submit query in part 2 of Figure 1 causes the generation, deployment and running of the cloud-orchestra – the execution of steps 3 to 5 in Figure 1. In this implementation we provide a means of listening by quickly starting up a jacktrip connection with the cloud, and connecting all of the synths connected to the DAC to the this connection. `teardown.sh` will shutdown the cloud-orchestra.

## 4.4 Experience with the cloud-orchestra

Connecting to the cloud-orchestra with jacktrip [6] results in relatively low latency. The overhead is not excessive and like `jack`, jacktrip was well made, with the concerns of the music community in mind. Streaming over websockets [10] increases latency considerably as any jitter or fluctations in the network can harm the latency of the TCP connection.

## 4.5 Recommendations

It is recommended [10] to maintain master and slave images that can be easily duplicated and deployed as instances (synth-cloud-orchestra includes a dockerfile). Debugging is often best done locally using docker (`http://docker.io`) and/or vagrant (`http://vagrantup.com`) to build local clouds. Locality tends to improve performance so network audio links should be avoided to reduce latency; star-network formations [6] for audio can flood switches and knock out local network abilities while tree formations do not. We emphasize the value of optimizing for locality as latency within VMs is less than latency between VMs.

# 5. CONCLUSIONS AND FUTURE WORK

Leveraging the cloud is difficult and leads to complicated orchestration and deployment. To address this difficulty we have described an abstract and concrete framework that enables computer musicians to define, create, and run cloud-orchestras. By leveraging model driven development one can allow end-user programmers, such as computer musicians, to focus on programming synthesizers rather than administering the cloud computers they rent. Model driven development allows musicians to avoid tedious manual configuration, and define their cloud-orchestra as a network of connected synthesizers and then deploy an optimized version of that network to existing resources.

Abstractly a method of assigning resources to a cloud-orchestra has been presented. Concretely, the example implementation *synth-cloud-orchestra* allows single-click deployment of cloud-orchestras defined by drag and drop. This work provides a novel user interface to design, define, deploy, and run cloud-orchestras.

This work opens up new areas for computer music performance related to cloud-orchestras, and the exploitation of cloud computing in computer music. One area of research includes optimal and near optimal resource allocation heuristics that could potentially optimize according to psychoacoustic models with respect to latency. The integration of other sources of information into the models is another area. Work should be done to create real-time/run-time configuration and re-configuration of the cloud-orchestra much like modifying a max/MSP or Pd patch live, except with numerous machines. OSC support should be included in the model as well. More work should be done on system liveness and awareness as well as enabling the distribution of user interfaces. Currently there is much work to be done on easing the provisioning of a cloud-orchestra and estimating the minimum resources needed to provision a cloud-orchestra. This current approach generates a static cloud-orchestra, further work could be done to generate dynamic and fail-safe cloud-orchestras. Synchronization of audio and control signals within the cloud should be investigated.

# 6. REFERENCES


[1] J. Allison. Distributed performance systems using html5 and rails. In *Proceedings of the 26th Annual Conference of the Society for Electro-Acoustic Music*, 2011.

[2] J. Allison, Y. Oh, and B. Taylor. Nexus: Collaborative performance for the masses, handling instrument interface distribution through the web. In *NIME*, 2013.

[3] Á. Barbosa. Displaced soundscapes: A survey of network systems for music and sonic art creation. *Leonardo Music Journal*, 13:53–59, 2003.

[4] S. D. Beck, S. Jha, B. Ullmer, C. Branton, and S. Maddineni. Grendl: Grid enabled distribution and control for laptop orchestras. In *ACM SIGGRAPH Posters*, 2010.

[5] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616, 2009.

[6] J.-P. Cáceres and C. Chafe. Jacktrip/soundwire meets server farm. *Computer Music Journal*, 34(3):29–34, 2010.

[7] A. Carôt, T. Hohn, and C. Werner. Netjack–remote music collaboration with electronic sequencers on the internet. In *Proceedings of the Linux Audio Conference*, 2009.

[8] L. Dahl, J. Herrera, and C. Wilkerson. Tweetdreams: Making music with the audience and the world using real-time twitter data. In *International Conference on New Interfaces For Musical Expression*, Oslo, Norway, 2011.

[9] A. V. Deursen, E. Visser, and J. Warmer. Model-driven software evolution: A research agenda. In *International Workshop on Model Driven Software Evolution*, 2007.

[10] A. Hindle. Cloudorch: A portable soundcard in the cloud. *Proceedings of New Interfaces for Musical Expression (NIME), London, United Kingdom*, 2014.

[11] S. Jordà. Multi-user Instruments: Models, Examples and Promises. In *NIME'05*, pages 23–26, 2005.

[12] S. W. Lee and G. Essl. Models and opportunities for networked live coding. *Live Coding and Collaboration Symposium 2014*, 1001:48109–2121, 2014.

[13] J. Oh and G. Wang. Audience-participation techniques based on social mobile computing. In *Proceedings of the International Computer Music Conference 2011 (ICMC 2011)*, Huddersfield, Kirkless, UK, 2011.

[14] S. Smallwood, D. Trueman, P. R. Cook, and G. Wang. Composing for laptop orchestra. *Computer Music Journal*, 32(1):9–25, 2008.

[15] D. Trueman. Why a laptop orchestra? *Organised Sound*, 12(02):171–179, 2007.

[16] P. Venezia. Review: Puppet vs. chef vs. ansible vs. salt. `http://www.infoworld.com/article/2609482/data-center/review--puppet-vs--chef-vs--ansible-vs--salt.html`, 2013.

[17] N. Weitzner, J. Freeman, S. Garrett, and Y.-L. Chen. massMobile - an Audience Participation Framework. In *NIME'12*, Ann Arbor, Michigan, May 21-23 2012.